An Efficient k-means Algorithm on CUDA

Jiadong Wu and Bo Hong
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, USA
jwu65, bohong@gatech.edu

***Abstract*—The k-means algorithm is widely used for unsupervised clustering. This paper describes an efficient CUDA-based k-means algorithm. Different from existing GPU-based k-means algorithms, our algorithm achieves better efficiency by utilizing the triangle inequality. Our algorithm explores the trade-off between load balance and memory access coalescing through data layout management. Because the effectiveness of the triangle inequity depends on the input data, we further propose a hybrid algorithm that adaptively determines whether to apply the triangle inequality. The efficiency of our algorithm is validated through extensive experiments, which demonstrate improved performance over existing CPU-based and CUDA-based k-means algorithms, in terms of both speed and scalability.**

***Keywords*-GPU; CUDA; k-means.**

I. Introduction

Clustering is an important technique with applications in many fields such as computer vision and genome analysis. The goal of clustering is to group a number of input data points into several sets, so that data points within one set share similar characteristics. The k-means algorithm [1], also known as Lloyd's algorithm [2], is a well known clustering method and widely used in various applications.

The k-means algorithm assigns each data point to a closest cluster. For a data point, its distance to a cluster is defined as its distance to the centroid of the cluster, where the centroid of a cluster is defined as the mean position of all the data points contained in the cluster. k-means algorithm uses an iterative approach. Initially, the positions of k clusters' centroids are randomly chosen. In a standard k-means iteration, each data point is labeled to the nearest cluster. After all the data points labeled, the centroid of each cluster is then be updated according to its data points. The labeling step and updating step are iterated until the labels do not change any more.

Improving the performance of the k-means algorithm has attracted a lot of research attentions. Among the large number of proposed methods, an important approaches is to use triangle inequalities to eliminate unnecessary distance calculations when searching for the nearest centroid. Algorithms based on this idea have been studied by various researchers. Considerable speedups were reported over the standard k-means algorithm [3]–[5].

Parallelizing the K-means algorithm has also received a lot of research attention. The Graphic Processing Unit (GPU) has recently emerged as a powerful parallel computing platform, and it has demonstrated significant speedups for many scientific applications. Researchers have studied k-means clustering on GPU [6]–[8]. All these existing works chose to implement the standard k-means algorithm because it can be easily load balanced (for every data point p, the standard k-means algorithm needs to calculate and compare p's distance to all the centroids, thus each data point bears the same amount of workload), and reasonable speedups have been observed. However, the performance of such existing GPU based k-means algorithms is affected by the inefficiency of the standard k-means algorithm, which, as demonstrated in [3]–[5], may perform significantly more distance calculations than necessary (which can be eliminated through triangle inequalities).

In this paper, we present a novel CUDA-based k-means algorithm with efficient support of the triangle inequality. We also present the technique of rearranging the data layout to explore the trade-offs between load balance and memory access coalescing. Because the reduction in the number of distance calculations (via the triangle inequality) depends on the characteristics of the input data set, we further present a hybrid algorithm to optimize the speed of k-means clustering on general input data set. The hybrid algorithm will adaptively determine whether to invoke the triangle inequality. The performance of the algorithms are verified through extensive experiments.

This paper makes the following major contributions:

1) we develop an efficient CUDA-based algorithm to accelerate k-means clustering by using triangle inequalities;
2) we design a data layout management technique to explore the trade-off between load balance and memory access coalescing;
3) we design an adaptive hybrid k-means algorithm for general input data set.

The rest of the paper is organized as follows: In Section II, a brief description of k-means algorithm is presented and related works are reviewed. In Section III, we present our triangle inequality reinforced k-means algorithm are studied.

1530-2075 2011
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/IPDPS.2011.331

IEEE computer society

Data layout management technique for memory access coalescing and load balancing is also presented. In Section IV, the limitations of the inequality reinforced algorithm are discussed. Our hybrid algorithm for speeding up k-means clustering on general input data sets is then proposed. In Section V, experimental results are presented to demonstrate the performances of the algorithms.

II. Background and Related Works

A. The k-means Algorithm

The k-means algorithm targets the following clustering problem: given a set of data points $\{P_{1...n}\}$, where each point is a d-dimensional vector, the problem is to partition the n points into k clusters $\{S_{1...k}\}$ ($k < n$) with centroids $\{C_{1...k}\}$ so as to minimize the in-cluster sum of squares of distances:

$$\arg\min_S \sum_{i=1}^{k} \sum_{P_j \in S_i} ||P_j - C_i||^2 \qquad (1)$$

The problem is NP-hard in Euclidean space for a general number of clusters k. The standard iterative k-means algorithm, as introduced in [1], [2], is actually a heuristic algorithm. The algorithm generally converges very fast but it does not guarantee global optimality. Each iteration of this algorithm consists of two steps: data point labeling and cluster centroid updating. The labels $\{L_{1...n}\}$ are notations for the assignment of data points to the clusters. For example, $L_j = i$ denotes that C_i is the nearest centroid to P_j, which also means that $P_j \in S_i$. The labeling step has $O(ndk)$ operations, in which each of the n points will be labeled to a cluster with nearest centroid by evaluating its distances to all the k centroids in d-dimensional space. Based on the labels, the updating step uses only $O(nd)$ operations to calculate the new centroids of all the k clusters. Computation wise, the labeling operations dominate the cost, especially when the number of clusters is large. Consequently, improving the labeling step has been the focus of many research efforts (as well as ours).

Accelerating k-means clustering algorithm by applying triangle inequalities in the labeling step [3]–[5] is based on the idea that many distance calculations in the labeling step may be redundant. For example, given a data point P_x and two clusters with centroids C_i and C_j. When labeling P_x, in the standard algorithm, two distance calculations are needed to compute $d(P_x, C_i)$ and $d(P_x, C_j)$. However, we can first compute $d(P_x, C_i)$, and if the inequality $d(C_j, C_i) > 2d(P_x, C_i)$ holds, then by triangle inequality, we can infer immediately that $d(P_x, C_j) > d(P_x, C_i)$ without computing the $d(P_x, C_j)$. Consequently, with all the inter-centroid distances calculated before the labeling step in each iteration, we can avoid up to $k - 1$ times of distance calculation in labeling each point, and reduce complexity of this step to $O(ndk')$ with $1 \leq k' \leq k$. The k' denotes the reduced number of distance calculations, and its value depends on characteristics of the input data set.

Algorithm 1 One iteration of CUDA-based standard k-means algorithm.

```
1: Copy $\{C_{1...k}\}$ to GPU device
2: Launch the GPU kernel to label $\{P_{1...n}\}$ to the nearest
   centroids
3: Copy $\{L_{1...n}\}$ back to host
4: Calculate the mean for each cluster and update $\{C_{1...k}\}$
```

B. Programming on CUDA

GPU has recently advanced from specialized fixed-function processor to highly programmable massively parallel computing device. With the help of general programming support from toolkits such as Compute Unified Device Architecture (CUDA), GPU is now exposed to programmers as a general-purpose shared-memory many-core computing platform, and has played important roles in many computation intensive applications.

GPU achieves massive parallelism through simultaneously launching of a large number of lightweight threads. The GPU threads are scheduled onto processor cores in the unit of a warp rather than of a single thread [9]. Different warps execute independently regardless of whether they are executing common or disjoint code paths. Threads within a warp execute the same instruction at a time but on different pieces of data, so maximum efficiency is achieved when all threads of a warp have identical execution path. If a warp of threads diverge on data-dependent conditional branches, the warp will sequentially execute all the branch paths, and subsequently nullify results from threads that are not on that path. The threads converge back to the same execution path after all paths are completed. Note that diverged execution paths often cause significant degradation to the efficiency of GPU programs. In summary, if an algorithm fits the Single Instruction Multiple Thread (SIMT) model (with few diverged execution paths), GPU can achieve substantial increase of computing performance.

As the standard k-means algorithm exhibits data parallelism (the labeling steps for the data points are identical and independent of each other), it can potentially benefit from the massively parallel GPU architectures. This feature had been explored even before the CUDA platform was introduced, researchers had studied the possibilities of doing k-means clustering on GPU with built-in functions in graphic library [10], [11]. With the help of CUDA, the implementation of standard k-means algorithm on GPU becomes easier and the performance is further improved [6]–[8]. Alg. 1 illustrates the the general CUDA algorithm of a standard k-means iteration with the input data points $\{P_{1...n}\}$ stored on GPU's global memory.

Algorithm 2 One iteration of CUDA-based inequality reinforced k-means algorithm.

```
1: Calculate inter-centroid distances ($ICD$) matrix
2: Sort each row of the $ICD$ matrix to derive the ranked
   index ($RID$) matrix
3: Copy $\{C_{1\dots k}\}$, $ICD$, and $RID$ to GPU device
4: Launch the GPU kernel to label $\{P_{1\dots n}\}$ to the nearest
   centroids with the help of $ICD$, $RID$, and triangle
   inequalities
5: Copy $\{L_{1\dots n}\}$ back to host
6: Calculate the mean for each cluster and update $\{C_{1\dots k}\}$
```

However, the standard algorithm may perform unnecessary distance calculations. Therefore, in contrast to previous papers that mainly focused on implementing the standard k-means algorithm , we target research on CUDA-based triangle inequality reinforced k-means algorithm. The reinforced algorithm features reduced distance calculations, but it also introduces a new set of problems: threads now have the diverged execution paths and irregular memory access pattens. In the next section, we will analyze these problems in detail and present an efficient CUDA-based k-means algorithm that addresses these issues.

III. CUDA-based inequality reinforced k-means algorithm

In this section, we present the triangle inequality reinforced k-means algorithm.

Alg. 2 illustrates one iteration of the CUDA-based triangle inequality reinforced k-means algorithm. Compared with the standard algorithm (shown in Alg. 1), two auxiliary steps (calculating and sorting the inter-centroid distances) are included before the labeling step. In the reinforced algorithm, ICD is a $k \times k$ matrix storing the distances between every two centroids. RID is another $k \times k$ matrix, where each row is a permutation of $1, 2, \cdots k$ representing the closeness of the distances from C_i to other centroids. For example, $RID_{2,3} = 5$ means that the 3rd closest centroid to C_2 is C_5. When labeling the points, redundant calculations can be avoided by looking up ICD and RID. In each iteration of Alg. 2, the calculation of ICD and RID cost $O(k^2 d)$ and $O(k^2 \log k)$ operations respectively. When $n \gg k$, the $O(ndk')$ labeling cost still dominates the entire computation.

For each data point P_x, the inequality reinforced labeling step is performed as follows: Suppose in previous iteration P_x was labeled to centroid i. To find the nearest centroid in this iteration, the labeling step will loop through the centroids following the sequence stored in the ith row of RID matrix. The loop will terminate once the first centroid C_j satisfying $d(C_j, C_i) > 2d(P_x, C_i)$ is found. Because the algorithm follows the sequence in RID, any centroid C_k beyond C_j satisfies $d(C_k, C_i) > d(C_j, C_i)$, which implies

Algorithm 3 GPU kernel of the inequality reinforced labeling step.

```
1: $t \leftarrow n/GridSize/BlockSize$
2: $s \leftarrow BlockID \times n/GridSize + ThreadID \times t$
3: for  $i = s + 1$ to $s + t$ do
4:     $oldCnt \leftarrow L[i]$
5:     $oldDist \leftarrow dist(P[i], C[oldCnt])$
6:     $newCnt \leftarrow oldCnt$
7:     for $j = 2$ to $k$ do
8:         $curCnt \leftarrow RID[oldCnt][j]$
9:         if $ICD[oldCnt][curCnt] > 2 \times oldDist$ then
10:            break
11:        end if
12:        $curDist \leftarrow dist(P[i], C[curCnt])$
13:        if $curDist < newDist$ then
14:            $newDist \leftarrow curDist$
15:            $newCnt \leftarrow curCnt$
16:        end if
17:    end for
18:    $L[i] \leftarrow newCnt$
19: end for
```

$d(C_k, C_i) > 2d(P_x, C_i)$. Consequently, by applying triangle inequality, we can omit calculations thereafter since P_x is at least closer to C_i than to C_k or beyond. We then label P_x to the nearest cluster.

Alg. 3 shows the detailed GPU kernel of the inequality reinforced labeling step. The work of labeling n data points are distributed to a total number of $GridSize \times BlockSize$ threads, so each thread is responsible for a subset of $t = \lceil n/GridSize/BlockSize \rceil$ points (the last thread may receive less points than other threads). In this kernel, $dist()$ calculates the Euclidean distance. For each point, $dist()$ is invoked k' ($1 \leq k' \leq k$) times, and the exact value of k' depends on how many such calculations can be eliminated via the triangle inequality.

As mentioned in Section II-B, a major problem encumbering the efficiency of a CUDA algorithm is intra-warp execution path divergence. According to the warp scheduling mechanism, all execution paths will be evaluated by the warp, and if a thread does not need a particular execution path, it will still sit through the execution (with results nullified) and waste resources. In our algorithm, the divergence is caused by imbalance in the workload, which can be represented by how many distance calculations need to be performed by a thread to label a data point. In Alg. 3, each thread works on a subset of data points. It is likely that for the data points that a warp of threads are working on, different numbers of distance calculations are needed, i.e. the threads may break from line 10 of the for loop (lines 7–17) at different iterations. Consequently, the overall efficiency of the warp is compromised because threads finishing earlier will still be executed as the slower threads are executed, only to see their results nullified. Consideration the scenario where a warp of 32 threads work on 32 data points, with one data point needing 10 distance calculations and the other 31 data points needing 1 calculation each, such path divergence will lead to an 87% loss in efficiency for the CUDA-based algorithm.

In addition to execution path divergence caused by workload imbalance, the overhead of memory access is also crucial for the efficiency of Alg. 3. Because $O(nd)$ space is needed to store the data points, which easily outgrows the capacity of shared memory for any meaningful problem sizes, we can only fit the input in the global memory, the slowest in the GPU memory hierarchy. Alg. 3 requires roughly the same number of global memory reads as the number of distance calculations. The overhead of global memory read will thus directly influence the performance. If these global memory reads can not be coalesced, the efficiency of the reinforced algorithm will be significantly impacted.

Unfortunately, load balance and memory coalescing are intertwined in the inequality reinforced k-means algorithm. Figure 1 gives an illustrative example of the problem. A warp of 32 threads, marked as T1 to T32, are working on a set of data points. Those data points are represented by the blocks in the figure. A row major two-dimensional array is assumed for the data layout of the points in GPU's global memory, where the first data point of thread T1 is followed by the first data point of T2, ..., the first data point of T32 is followed by the second data point of T1, and so on. The number marked within each block indicates the workload (number of distance calculations) needed to label this point. The shaded blocks indicate the points on which the warp of threads are working currently. A straightforward choice is to process the points row by row as shown in the left sub-figure. This will cause all the memory accesses to be coalesced, but the warp will suffer from the problem of execution path divergence. Alternatively, if the number of distance calculations for each point is known, the threads can process the data points in the decreasing order of their workloads as is shown in the right sub-figure. The ordered processing will reduce execution path divergences, but will unfortunately cause significant overheads because memory accesses are not coalesced.

The problem of unbalanced workload and the problem of uncoalesced global memory access can not be solved simultaneously unless the data layout in the memory matches the order that data points are processed, and such an order should minimize execution path divergence. Based on these considerations, we designed a novel CUDA-based k-means algorithm as illustrated in Alg. 4. The key ideas are to rearrange the processing order according to the workload needed by each data point, and to rearrange the data layout in GPU's global memory to match the processing order.

To implement these ideas, we break the inequality reinforced algorithm into two epochs. Given a set of input data points $\{P_{1...n}\}$, let $\{M_{1...n}\}$ denote the number of distance calculations done for each point in certain k-means iteration. $\sum_{i=1}^{n} M_i/n$ represents the average number of distance calculations for one data point in one k-means iteration, and this value will change iteration by iteration. In Epoch 1 (early stages of the algorithm), Alg. 3 is applied and $\{M_{1...n}\}$ is monitored. When $\sum_{i=1}^{n} M_i/n$ converges (with respect to a preset threshold), the algorithm exits Epoch 1 and enters Epoch 2. Epoch 2 will access the data points in a different order than than Epoch 1. In Epoch 2, the algorithm processes the data points $\{P_{1...n}\}$ in a rearranged order $\{P_{a_1...a_n}\}$ where $\{a_{1...n}\}$ is a permutation of $1...n$ such that $M_{a_i} \geq M_{a_{i+1}}$. To coalesce memory accesses, the linear storage of $\{P_{1...n}\}$ will also be rearranged to $\{P_{a_1...a_n}\}$ before Epoch 2 starts. The data layout therefore matches the processing order of the data points for Epoch 2, and the memory accesses by the threads will subsequently be coalesced. The termination condition of this algorithm is the same as of the standard k-means algorithm.

In the following analysis, we will discuss the efficiency and correctness of such a two-epoch algorithm.

The fundamental assumption we used in our algorithm is that the workload of a data point would be somewhat

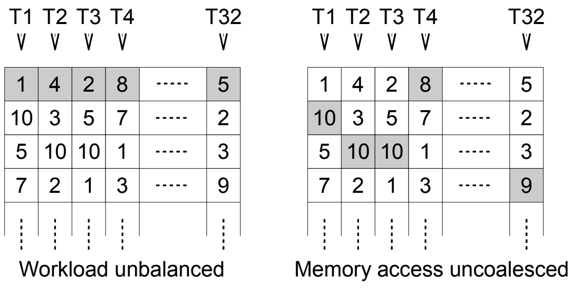


Figure 1. The problems impeding the efficiency of inequality reinforced k-means algorithm on CUDA.

Algorithm 4 Efficient CUDA-based inequality reinforced k-means algorithm.

```
1: Initialize $\{C_{1...k}\}$
2: Copy $\{P_{1...n}\}$ to GPU device
3: **repeat**
4:     Alg. 2, and recording $\{M_{1...n}\}$
5: **until** $\sum_{i=1}^{n} M_i/n$ converge
6: Rearrange $\{P_{1...n}\}$ to $\{P_{a_1...a_n}\}$ s.t. $\{M_{a_1...a_n}\}$ is in decreasing order
7: Copy $\{P_{a_1...a_n}\}$ to GPU device
8: **repeat**
9:     Alg. 2
10: **until** clusters converge
```

stationary through the k-means iterations. By examining the computation history of many k-means iterations in our experiments, we noticed that, although a data point P_i may need different numbers of distance calculations M_i in different k-means iterations, M_i will stabilize after a few iterations, because M_i is determined only by the relative position of P_i to the centroids. Data points near the centroid of one cluster require less distance calculations as many of them can be eliminated by the triangle inequalities; data points near the boarder of clusters require more distance calculations as they may have similar distances to multiple centroids (thus rendering the triangle inequalities less effective). It is typical in k-means clustering that the centroids change dramatically in the first few iterations and then stabilize (with minor oscillations) in later iterations. When the centroids stabilize, their relative positions to the points are also fixed. Thus, for a given data point P_i, the value of M_i is expected to stabilize or fluctuate only slightly after a few iterations.

Figure 2 presents an example of such converging behavior. We generated ten 245760-point 32-dimensional input data sets[1] and then clustered each of them into 32 clusters. The data sets are generated to exhibit different reductions in distance calculations. In the figure, each curve depicts the average value $\sum_{i=1}^{n} M_i/n$ for one input data set, from the 1st to the 25th k-means iteration. We can see that for every data set the value of $\sum_{i=1}^{n} M_i/n$ drops and then converges. Given the observation that such data set generally needs more than 100 k-means iterations to terminate, the average value of M converges much earlier than the clusters could. In a larger scale experiment, we tested 240 input sets of similar size. To cluster those sets an average of 130.15 iterations are needed. If the convergence criteria is set as that variation between two consecutive iteration is less than 1%, then an average of only 4.63 iterations are needed for $\sum_{i=1}^{n} M_i/n$ to converge. It is reasonable to extrapolate that such convergence behavior is typical in k-means clustering problems.

With this converging behavior of workload, for any data point P_i, the value of M_i at the end of Epoch 1 can be used as estimation of M_i in the iterations thereafter. After Epoch 1, we rearrange the processing sequence of the data points such that $\{P_{1...n}\}$ are processed in the decreasing order of their M values. It can be proved that such a processing sequence guarantees optimal workload balancing. This is formally stated below.

Given data points $\{P_{1...n}\}$ and the related $\{M_{1...n}\}$, suppose the points are processed by a warp of w threads in the order of $\{P_{x_1...x_n}\}$, where $\{x_{1...n}\}$ is a permutation

[1]The input sets are from group A of our experiments, and the details can be found in Section V-A.

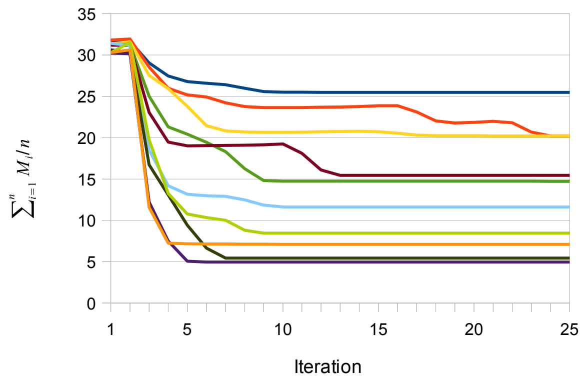


Figure 2. The converging behavior of $\sum_{i=1}^{n} M_i/n$. Each line represents experiment on one data set.

of $\{1...n\}$. We can define the imbalance ratio as

$$R_x = \frac{\sum_{i=1}^{n/w} \max\{M_{x_{(i-1)w+1}}...M_{x_{(i-1)w+w}}\}w}{\sum_{i=1}^{n} M_i} - 1, \quad (2)$$

which reflects how much execution time is wasted due to divergence in the execution paths. The ratio reaches its theoretical lower bound 0 if all the threads have exactly the same amount of workload.

Lemma 1. *The imbalance ratio will not be affected by the processing order of data points within a warp, or by the order that warps are scheduled.*

Proof: By definition,

$$\max\{M_{x_{(i-1)w+1}}...M_{x_{(i-1)w+w}}\}$$

determines the effective workload of each warp i after warp scheduling is considered, and the summation is independent of the processing order within a warp or how warps are scheduled. □

Theorem 1. *For any given set of data points, processing the points in decreasing order of their workloads minimizes the imbalance ratio.*

Proof: Given data points $\{P_{1...n}\}$ and the sequence $\{P_{a_1...a_n}\}$ of these points in decreasing order of the M values, with

$$M_{a_i} \geq M_{a_j}, \forall 1 \leq i < j \leq n. \quad (3)$$

The imbalance ratio of processing $\{P_{a_1...a_n}\}$ on GPU is

$$R_a = \frac{\sum_{i=1}^{n/w} M_{a_{(i-1)w+1}} w}{\sum_{i=1}^{n} M_i} - 1. \quad (4)$$

We will show that R_a is optimal.

For the sake of contradiction, suppose for the same data set, the optimal imbalance ratio is achieved by another processing order $\{P_{b_1 \dots b_n}\}$, which yields an imbalance ratio R_b and $R_b < R_a$. According to Lemma 1, we can construct another optimal sequence $\{P_{c_1 \dots c_n}\}$ by rearranging $\{P_{b_1 \dots b_n}\}$ as follows. First sort the points inside each warp i in decreasing order of M to make

$$M_{c_{(i-1)w+j}} \geq M_{c_{(i-1)w+k}}, \forall 1 \leq j < k \leq w; \qquad (5)$$

and then sort the warps in decreasing order of each warp's largest M, i.e.

$$M_{c_{(i-1)w+1}} \geq M_{c_{(i)w+1}}, \forall 1 \leq i < n/w. \qquad (6)$$

such that,

$$R_c = \frac{\sum_{i=1}^{n/w} M_{c_{(i-1)w+1}} w}{\sum_{i=1}^{n} M_i} - 1 = R_b < R_a. \qquad (7)$$

Consequently, there exists at least a pair of ith warps in $\{P_{a_1 \dots a_n}\}$ and in $\{P_{c_1 \dots c_n}\}$ that satisfies

$$M_{a_{(i-1)w+1}} > M_{c_{(i-1)w+1}}, \qquad (8)$$

which leads to a contradiction. Because, according to inequality (3), $M_{a_{(i-1)w+1}}$ is larger than that of at most $n-(i-1)w-1$ points, but according to inequality (5) and (6), $M_{c_{(i-1)w+1}}$ is larger than that of at least $n-(i-1)w-1$ points. Such contradiction indicates that no sequence can yield a smaller imbalance ratio than R_a. $\square$

As discussed previously, the layout of the data points are rearranged after Epoch 1, and the algorithm will use data in the new layout for Epoch 2 until the algorithm converges. During Epoch 2, the data layout on global memory is identical to the rearranged processing order. Thus threads within a warp will request to access a continuous region of the global memory, thereby achieving coalesced memory accesses. The extra cost of rearranging the data and transferring them to GPU is $O(n \log n) + O(nd)$, which is acceptable because it happens only once in the entire clustering process. Due to the fact that majority of the iterations run on the rearranged data, memory accesses are expected to be efficient with our algorithm.

Figure 3 demonstrates the performance of Alg. 4 on a group of 240 input data sets[2]. In Figure 3, 'Theoretical upper bound' denotes the performance of applying triangle inequality without considering the impact of warp scheduling, i.e. it denotes the average value of $1 - \sum_{i=1}^{n} M_i/n/k$ over all iterations. Due to the cycles wasted in the warp scheduling mechanism, the percentage of calculation saved by the reinforced CUDA algorithm, which is the average $\sum_{i=1}^{n/w} \max\{M_{x_{(i-1)w+1}} \dots M_{x_{(i-1)w+w}}\} w/n/k$ over all iterations, will be smaller than the theoretical upper bound. In this experiment, the reinforced algorithm is capable of saving up to 78% of calculations compared to the standard k-means algorithm. When the reinforced algorithm running on

[2]The input sets are also from group A of our experiments with details discussed in Section V-A.

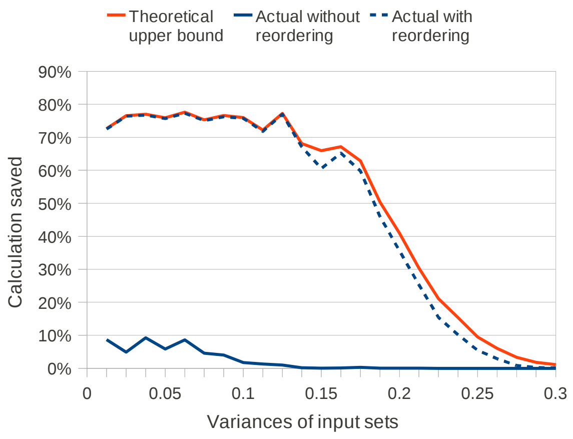


Figure 3. The efficiency in reducing the number of distance calculations.

GPU without rearranging the processing order, only a small percentage of calculations can be saved. This is because in such cases lots of cycles are wasted due to load imbalance and execution path divergence. However, when the algorithm runs with our rearranged processing order, the reduction in calculations is very close to the theoretical upper bound.

IV. Efficient Hybrid algorithm for Generic k-means Clustering

The effectiveness of the reinforced algorithm is sensitive to the input data. Three factors contribute to the sensitivity. First, the complexity of labeling step in the reinforced k-means algorithm is $O(ndk')$ with k' between 1 and k (depending on the reduction via triangle inequalities). The efficiency of the algorithm will be compromised if k' is comparable to k (i.e. when most data points need to calculate distances to most data centroids for the labeling step). Under this situation, the cost of data rearrangement and determining triangle inequalities may not be justifiable. The second situation is limited shared memory size. In our algorithm, the information of centroids is stored on the shard memory, it needs exactly $4kd$ bytes space for single-precision data. As k and d scale up, the GPU may run out of shared memory. In such cases, information of centroids can only be stored on the global memory. Because the accessing pattern of this data is random, overhead will be introduced. The third factor emerges when k is too large, thus the $O(k^2 \log k)$ cost of calculating RID dominates. To avoid such inefficiencies, we develop the following hybrid algorithm to handle general input data.

In the hybrid algorithm, whether to start the reinforced algorithm is decided in the initialization stage. If k is larger than a threshold, the algorithm will use the standard k-means kernel. Otherwise, the reinforced k-means kernel will be chosen. For the reinforced k-means algorithm, a second decision as to whether the reinforced algorithm is worth

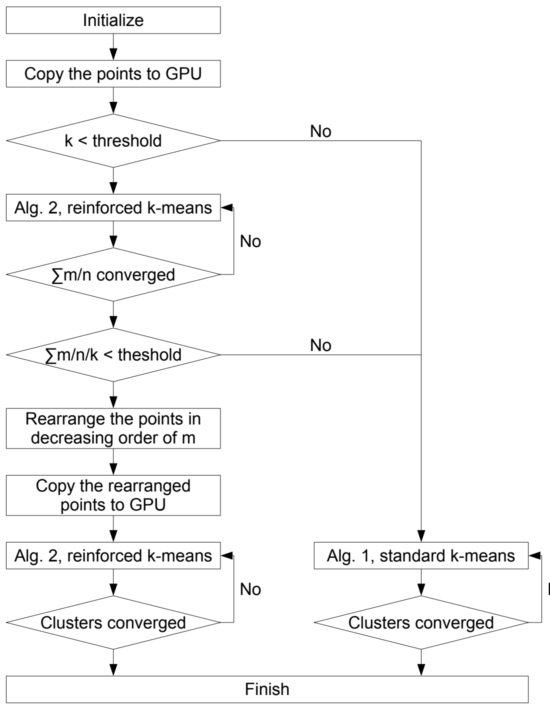


Figure 4. The CUDA-based hybrid k-means algorithm.

continuing in Epoch 2, is made once the values of M stabilize after certain initial iterations. The summation of M will be examined before data rearrangement. If the ratio of $\sum_{i=1}^{n} M_i$ to nk exceeds a certain threshold, the algorithm predicts that not enough operations can be saved in the future through triangle inequalities. In such cases, maintaining the matrices of ICD and RID will cause extra overhead and increase the execution time; and the algorithm should use the kernel for standard k-means instead of reinforced k-means to finish the following iterations. The proposed hybrid algorithm, listed in Figure 4, is therefore capable of adaptively maximizing the speed of k-means clustering according to the characteristics of the input data set.

In order to determine the appropriate thresholds, we compare the cost of each k-means iteration, which is $andk' + bk^2 \log k$ for the reinforced algorithm, and $cndk$ for the standard algorithm, where a, b, c denotes the constant factors in the asymptotic complexity bound. In general, the cost of reinforced algorithm is smaller than that of the standard algorithm when k' is small, and it will be greatly increased if $bk^2 \log k$ is comparable to $cndk$. We exam the value of $k \log k$ versus cnd/b. If $k \log k > cnd/b$, the algorithm will launch the standard k-means kernel.

If the $k \log k < cnd/b$, then the reinforced k-means kernel will be launched, and the ratio of $k'/k = \sum_{i=1}^{n} M_i/n/k$ will be tested once $\sum_{i=1}^{n} M_i/n$ converges. According to the model, the costs of two algorithms are equal when k' is large enough to satisfy $andk' + bk^2 \log k = cndk$. Thus the threshold can be calculate as:

$$\frac{k'|_{andk'+bk^2 \log k = cndk}}{k} = \frac{c}{a} - \frac{bk \log k}{adn} \tag{9}$$

In these thresholds, c/a and b/a are characteristics of the implementation as well as the hardware platform, and are independent from the input points. The values can be obtained through experiments.

V. Experiments and Results

A series of input data sets are generated to evaluate the algorithms. For each input set, k uniform random initial centroids are first placed inside a d-dimensional hyper-cube of side length 1. n/k points are then placed around each centroid with Gaussian distribution $(0, \sigma^2)$. By altering the standard variance σ^2 from 0.001 to 1, different input sets of size $n \times d$ can be generated.

All these experiments are conducted on a machine equipped with an Intel Xeon E5520 CPU and a Nvidia Tesla C1060 GPU. 2.6.18 x86_64 Linux kernel was used with gcc 4.1.2 and CUDA 3.10. In the CUDA algorithms, the grid size and block size are set as 30 and 256. All euclidean distance calculations are done in single-precision.

The major indicator we used to evaluate the performance of algorithms is $EALT$, the Equivalent Average Labeling Time.

$$EALT = \frac{total\ execution\ time - updating\ time}{number\ of\ iterations} \tag{10}$$

In the evaluations, EALT serves as a better indicator than the total execution time for two reasons. First, the total number of k-means iterations varies significantly on different input data sets and on different initial points, so it is less meaningful to compare the total execution times. Second, our research focuses on the labeling of data points (which dominates the execution time of k-means clustering). The procedure of updating the centroids is the same for all the algorithms mentioned in this paper. For the standard algorithm, $EALT$ is the same as average labeling time of each k-means iteration. For the reinforced algorithm and hybrid algorithms, all the extra overheads, such as rearranging points and computing the ICD RID matrices, are amortized into $EALT$ in addition to the labeling time.

A. Adaptation to inputs with different σ^2

The first experiment is to demonstrate the performance of the hybrid CUDA algorithm and its adaptation to different input sets. With our procedure of generating the input data sets, σ^2 value will affect how many calculations can be saved by the inequality reinforced algorithm. If σ^2 is large, points will be more scattered, and the reinforced algorithm is less likely to save calculation. If σ^2 is small, points are located

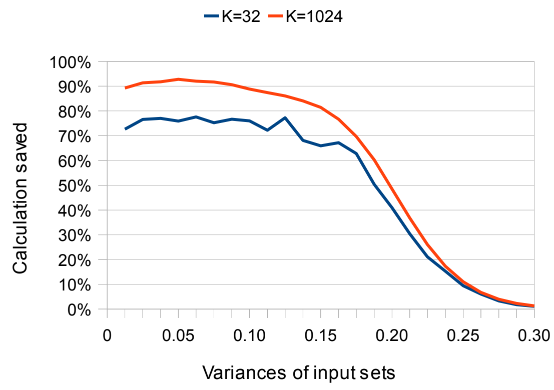


Figure 5. The equivalent percentage of calculation saved by reinforced algorithms.

closer to the centroids of the clusters, and the reinforced algorithm will be more efficient.

Two groups of input sets were used. Group A consists of 240 input sets each with 245760 points in 32 dimensions around 32 initial centroids. The 240 sets are generated with 24 different σ^2 values (10 sets each) ranging linearly from 0.0125 to 0.3. Group B also consists of 240 input sets generated in same setting except that the number of the centroids is 1024. On these two groups, the relations between equivalent percentage of calculation saved by reinforced algorithm and the σ^2 of inputs are illustrated in Figure 5. For inputs with smaller σ^2, up to 78% of calculation can be saved, but for inputs with σ^2 of about 0.3 (or larger), no calculation can be saved.

On the two groups of inputs, five kinds of k-means clustering algorithms are tested. These are 1) Standard CPU algorithm, 2) Reinforced CPU algorithm, 3) Standard CUDA algorithm, 4) Reinforced CUDA algorithm, and 5) Hybrid CUDA algorithm. The two CPU algorithms used for comparison are single threaded algorithms. The results are listed in Figure 6.

Compared with CPU-basd k-means algorithms, the hybrid CUDA algorithm achieved up to 75X speedup over standard CPU algorithm and up to 21X speedup over reinforced CPU algorithm at $k = 32$. In the case of $k = 1024$, it achieved speedups of up to 33X and 12X respectively. Compared with other two CUDA algorithms, the hybrid algorithm is able to adapt to the input data sets and consistently achieve the same or better performance. The reinforced CUDA algorithm is very efficient when inputs has small σ^2 and when k is small, in these cases the hybrid algorithm exhibits the same level of efficiency.

Table I
THE COMPOSITION OF $EALT$.

	Reinforced CPU alg.		Hybrid CUDA alg.	
# of clusters	32	1024	32	1024
Labeling	99.98%	97.75%	94.14%	87.94%
Computing ICD	0.02%	0.59%	0.36%	3.06%
Computing RID	0.03%	1.66%	0.61%	7.90%
Hybridizing			4.92%	1.11%

B. Decomposition of $EALT$

We decomposed $EALT$ of the reinforced CPU algorithm and of the hybrid CUDA algorithm into four components: 1) Labeling the data points; 2) Computing inter-centroid distances to generate the ICD matrix; 3) Sorting inter-centroid distances to generate the RID matrix; 4) Hybridizing of the algorithm, including monitoring $\sum_{i=1}^{n} M_i/n$ and rearranging the data layout.

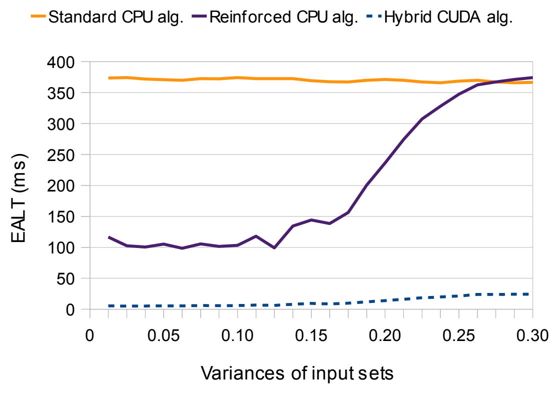
The decomposition is averaged over all the test data sets and the summary is listed in Table I, which shows that the labeling step dominates. The overhead of hybridizing and computation of RID contribute to only small portions of $EALT$. And the impact of other overheads are almost negligible in our algorithms.

C. Scalability of the algorithm

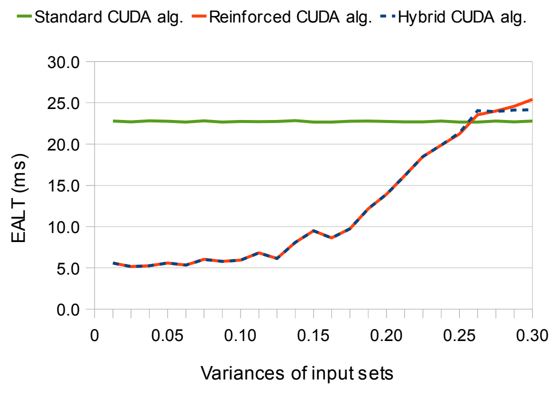
In actual k-means clustering problems, all the three parameters, n, d, and k may scale up. Scalability with respect to k requires special consideration, since the inequality reinforced algorithm as well as our hybrid CUDA algorithm is input sensitive for how large portion of k can be reduced. In our experiments, the scalability will vary depending on the value of σ^2.

In this experiment, three groups of input sets are generated to test the algorithms' scalability to k. Group C consists of 50 input sets each with 245760 points in 32 dimensions and a σ^2 of 0.15. The 50 sets are generated with 5 initial centroids number 64, 128, 256, 512, and 1024 each for 10 sets. Group D and E are similar to Group C except that σ^2 is 0.20 for Group D and 0.25 for Group E. The results are illustrated in Figure 7.

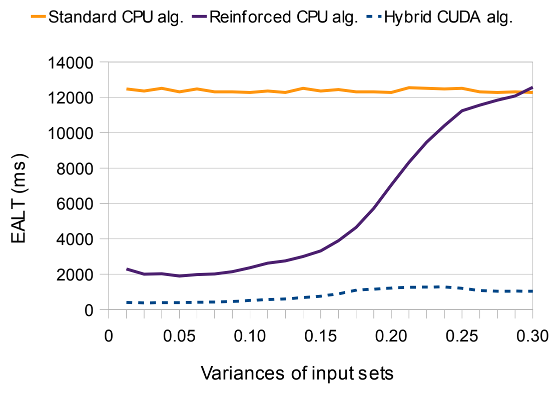
Although the standard CUDA algorithm is significantly less efficient on some inputs, it always exhibits better scalability to the number of clusters. The major reason, as mentioned in Section IV, is that inequality reinforced algorithm makes irregular accesses to the centroids' information. For a relatively small number of clusters, the information of centroids can be stored on shared memory, thus minimizing the influence of irregular memory accesses. However, in this scalability experiment, the information of centroids is stored on the global memory. So, these irregular accesses cause considerable overhead for uncoalesced global memory reads, and as k increases the overhead increases. To avoid this low efficiency situations, the hybrid CUDA algorithm switches



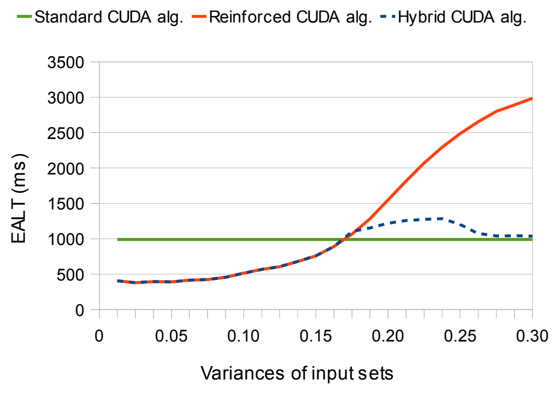
(a) Hybrid CUDA Alg. VS CPU Alg. on Group A



(b) Hybrid CUDA Alg. VS Other CUDA Alg. on Group A



(c) Hybrid CUDA Alg. VS CPU Alg. on Group B



(d) Hybrid CUDA Alg. VS Other CUDA Alg. on Group B

Figure 6. Performance comparison of k-means algorithms.

to the standard algorithm when k becomes large. Although the hybridization has also caused some overheads, the performance is very close to the standard CUDA algorithm at large k, and achieved better scalability than the reinforced CUDA algorithm.

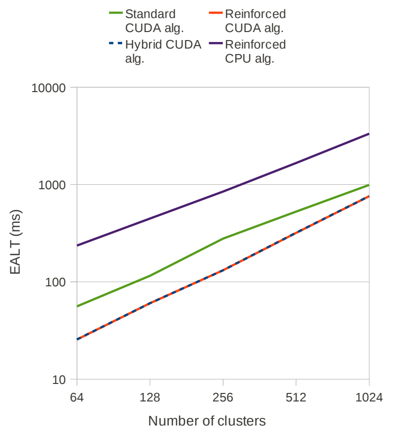
According to the analysis of complexity, the scalability of the algorithms to n and to d are simpler than the scalability to k. The only hard limitations are the size of shared memory and global memory on the GPU. Figure 8 shows the scalability with respect to n. The experiment is conducted on input group F, which consists of 50 input sets with d=32, k=32, $\sigma^2 = 0.15$, and n from 61440 to 983040. The results show that, all the k-means algorithms scales and the hybrid k-means algorithm outperforms others in terms of absolute execution speeds.

In summary, our CUDA-based reinforced k-means algorithm and the load-balancing technique substantially enhance the performance of the k-means clustering. The hybrid algorithm, with the novel idea of adaptive kernel selection, exhibits good performance in terms of both scalability and absolute execution speeds.

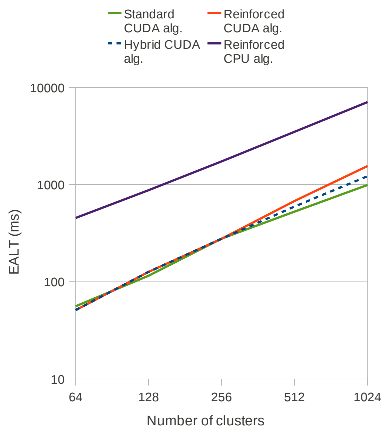
VI. Conclusion and Discussions

In this paper, we presented an efficient CUDA-based reinforced algorithm for k-means clustering with load balancing. We further developed a novel hybrid algorithm for k-means problems with arbitrary input data sets. The experiments demonstrated that our hybrid algorithm can improve the CPU-based single-threaded standard k-means algorithm by up to 75x.

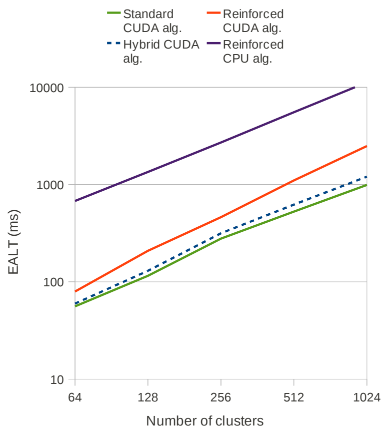
There are several potential directions to further improve the performance of k-means clustering on GPU. Currently, only the labeling step is accelerated by GPU. It may be beneficial to accelerate other steps such as calculating and sorting the inter-centroid distances as well as the centroid updating step. On the other hand, improving the irregular access of centroids' information may also be worth exploring.



(a) Group C



(b) Group D



(c) Group E

Figure 7. Scalability with respect to k.

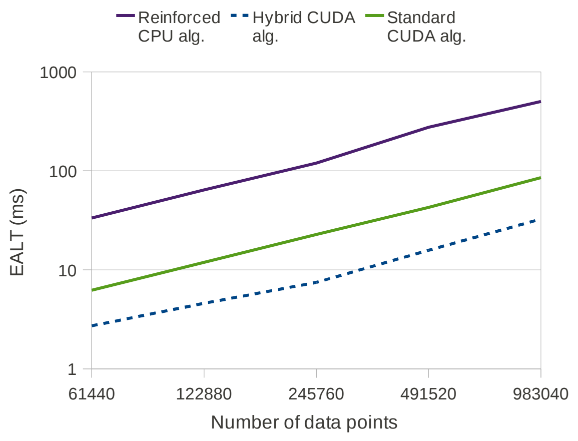


Figure 8. The scalability of algorithms to n the number of points.

ACKNOWLEDGMENT

This work is supported by the US National Science Foundation under award number CNS-0845583.

REFERENCES

[1] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds., vol. 1. University of California Press, 1967, pp. 281–297.

[2] S. Lloyd, "Least squares quantization in PCM," *Information Theory, IEEE Transactions on*, vol. 28, no. 2, pp. 129–137, Mar. 1982.

[3] M. E. Hodgson, "Reducing the computational requirements of the minimum-distance classifier," *Remote Sensing of Environment*, vol. 25, no. 1, pp. 117 – 128, 1988.

[4] M. Orchard, "A fast nearest-neighbor search algorithm," apr. 1991, pp. 2297 –2300 vol.4.

[5] S. J. Phillips, "Acceleration of k-means and related clustering algorithms," in *ALENEX '02: Revised Papers from the 4th International Workshop on Algorithm Engineering and Experiments*. London, UK: Springer-Verlag, 2002, pp. 166–177.

[6] S. Che, J. Meng, J. W. Sheaffer, and K. Skadron, "A performance study of general purpose applications on graphics processors," in *FIRST WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS*, 2007.

[7] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via CUDA," apr. 2009, pp. 7 –15.

[8] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell, "A parallel implementation of k-means clustering on GPUs," in *WorldComp 2008*, Las Vegas, Nevada, July 2008 2008.

[9] NVIDA, "CUDA C programming guide," Version 3.2, http://developer.nvidia.com.

[10] J. D. Hall and J. C. Hart, "Abstract GPU acceleration of iterative clustering," 2004.

[11] C. N. Vasconcelos, A. Sá, P. C. Carvalho, and M. Gattass, "Lloyd's algorithm on GPU," in *ISVC '08: Proceedings of the 4th International Symposium on Advances in Visual Computing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 953–964.