CPU-GPU Dynamic Approximation Project

Xi Chen, Mingze Gao, Yanzhou Liu
{xchen128, mgao1, yzliu}@umd.edu

I. Introduction

Approximate computing is an emerging design technique that leverages the inherent error resilience of applications for improved efficiency. For these applications, there is usually no specific "golden" output value that must be computed. Consequently, it's possible to relax the numerical equivalence between the specification and implementation of such applications. Examples of such applications include digital signal processing, image, audio, and video processing, graphics, web search, data analytics, etc.

A feature of these applications is that they have plenty of data level parallelism and the data can be processed independently and in any order on different processing elements for a similar set of operations such as filtering, aggregating, ranking, etc. General-purpose GPU (GPGPU) is deemed suitable for this new class of applications. GPGPU is capable of running many types of applications and has recently provided multiple cores to process data in parallel.

In our work, we focus on applying approximate computing on iterative methods and propose two lightweight quality estimators to effectively capture the computation quality in each iteration of IMs. We implemented two clustering algorithms, K-means algorithm and Expectation-Maximization (EM) algorithm using OpenCL programming language on NVIDIA's GPU. Clustering is to divide up data into groups/clusters so that points in the same group are more "similar" to each other in a way than points in other groups. Clustering algorithm is applied everywhere, such as marketing, classifications in biology and geology, insurance field, online documentation classification, etc.

II. Quality Evaluation Metric

In our project, we present two approximation criteria specific to clustering algorithm to monitor output quality. We use static design to replace fully accurate arithmetic components with approximate units to achieve various benefits with slight quality degradation.

In our first criterion, the quality evaluation metric is hamming distance. First, we need to know the largest distance between two points in the dataset. When we update current centers and compare them with the previous centers, if the sum of distance that all the centers move is less than 0.02% of the largest distance, we will consider this result as acceptable result and stop the iteration. The percentage 0.02% is user-defined value. When we set the criterion to 0.02%, we could achieve significant speedup with sacrificing clustering accuracy.

In the second criterion, we choose the cluster attribution of each node as the quality evaluation metric. When less than 5% of all the nodes change their cluster during the iteration, we will stop the iteration and output the approximate result. We also tested the case when less than 1% of all the nodes change their cluster in the iteration.

III. Block Diagram

Fig. 1 and Fig.2 are the block diagrams for K-means and EM algorithm.

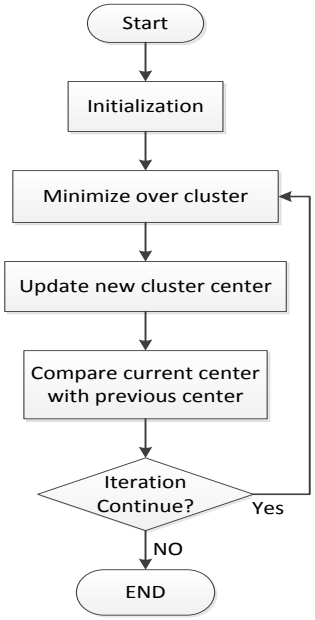

Fig 1. K-means Block Diagram

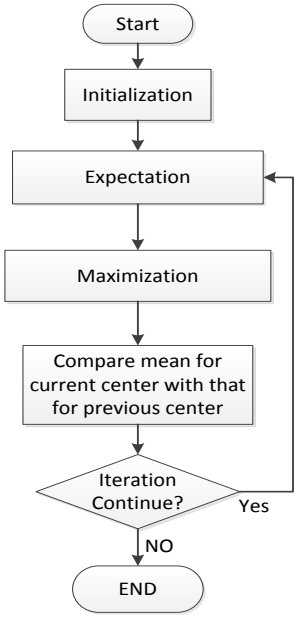

Fig 2. EM Block Diagram

K-means algorithm, known as Lloyd's algorithm is used to solve the clustering problem and works as follows in CPU-GPU platform. First, initialize the center of the clusters in CPU. In this step, CPU will load data and randomly select the cluster centers. Then attribute the closest cluster to each data point and set the position of center of each cluster to the mean of all data points belonging to that cluster. After that, we should compare current centers with previous centers. The computation process after initialization is performed in GPU. GPU will send the computational result back to CPU and CPU will decide whether the iteration should continue based on the criteria we talked about in section 2. If the iteration should continue, then we will repeat the iteration part. Otherwise, the result is considered to converge and we will output the result.

In statistics, the Expectation-Maximization algorithm is an iterative method for finding maximum likelihood estimates of parameters in statistical models, where the model depends on unobserved latent variables. EM is frequently used for data clustering in machine learning and computer vision. Since in our project, we mainly focus on the implementation, the principles behind EM algorithm are omitted.

Gaussian Mixture Model (GMM) is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think K-means is a particular case of EM algorithm.

Comparison to K-means clustering:
When implementing k-means algorithm, the points are assigned to the nearest cluster using the straight distance metric, for example the Euclidean distance. The Euclidean distance is a very straightforward metric which is less useful when the cluster contains significant covariance. The reason is shown in the following two figures. In figure 3, the red x and green x are equidistant from the cluster mean using the Euclidean distance, but intuitively we can see that the red x doesn't match the statistics of this cluster near as well as the green x. If we normalize and whitening the data to remove the covariance, we can clearly see that, in figure 4, the red x becomes much further than green x to the center.

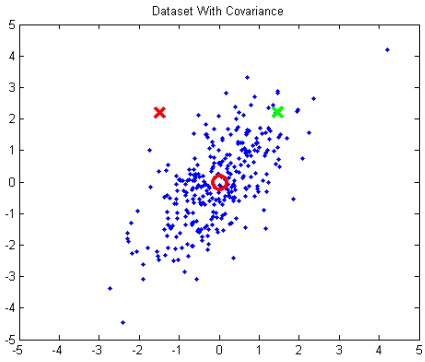

Fig 3. Dataset with covariance

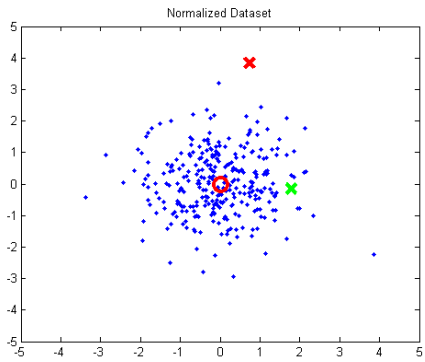

Fig 4. Normalized Dataset

Initialization is still in CPU. We will load data and set covariance matrix for clusters. Then in expectation step, we need to calculate the probability that each data point belongs to each cluster. In maximization step, re-calculate the cluster means and covariance based on the probabilities calculated in the expectation step. After that, compare means of the previous center with the current center. The computation process after initialization is performed in GPU, the same as K-means algorithm. Then GPU will send the result back to CPU to ask CPU if it should continue the iteration. If it should continue, repeat the computation and comparison process. Otherwise, stop iteration and output the result.

IV. Implementation Details

A. K-means algorithm

Computation on GPU for K-means algorithm is explained as follows:

● GPU side: (One item indicates one kernel function)

(1). Find the cluster center c_k which is closest to the node X_i; an array C will be generated where C(i) is the cluster center for node X_i.

(2). Scatter the array C(i) to Cd_{N*K} where Cd(i,j) indicates whether node i belongs to cluster j or not, so the elements in Cd will be 1 or 0. Cd(i,j)=1 denotes that the node i is in cluster j. The reason why we generate such matrix in the intermediate step is to reduce the computation in the follow steps. When updating the center, we only need to do the two matrixes multiplication.

(3). Minimize over the cluster centers c: the updated center location is the average of all nodes in that cluster; and compute difference between the updated cluster and current cluster.

● CPU side:

In a while loop, before GPU computation, initialize the cluster centers' locations first. After initialization, launch the kernels listed above in order and repeat until the centers converge.

As we observed, in the last few interactions, nodes in every cluster change a little. Therefore, it is possible to do some approximation for GPU algorithm with acceptable error. The criteria for iterations execution could be replaced by the difference lower than some threshold. This means the user may tolerate some approximate clustering instead of spending long time to achieve the accurate result. The value of threshold is determined through the Convex Optimization method using K-means function as a target function. Our designed run-time quality estimator will find the pareto points

and balance the result quality and performance (or energy consumption for limited resource devices). So we can call our design as a QoS-aware system.

B. EM algorithm

During initialization, before the expectation step, we

1) randomly select data points as the initial centers.
2) set the covariance matrix for each cluster to be equal to the covariance to the dataset.
3) set the "prior probability", which is the fraction of the dataset that belongs to each cluster.

In Expectation step, we will calculate the probability that each data point belongs to each cluster (using our current estimated means vectors and covariance matrices).
The main equations of E step are listed:

$$g_j(x) = \frac{1}{\sqrt{(2\pi)^n |\Sigma_j|}} e^{-\frac{1}{2}(x-\mu_j)^T \Sigma_j^{-1}(x-\mu_j)}$$

$$w_j^{(i)} = \frac{g_j(x)\phi_j}{\sum_{l=1}^{k} g_l(x)\phi_l}$$

Symbol	Meaning
$g_j(x)$	The PDF of the multivariate Gaussian for cluster j; th probability of this Gaussian producing the input x
j	Cluster number
x	The input vector (a column vector)
n	The input vector length
Σ_j	The n x n covariance matrix for cluster j
$\lvert\Sigma_j\rvert$	The determinant of the covariance matrix
Σ_j^{-1}	The inverse of the covariance matrix

Symbol	Meaning
$w_j^{(i)}$	The probability that example i belongs to cluster j
$g_j(x)$	The multivariate Gaussian for cluster j
ϕ_j	The "prior probability" of cluster j (the fraction of the dataset belonging to cluster j)
k	The number of clusters

Majority of the calculation in the first equation can be full paralleled.

1) Do the transpose of (X- μ_j). The size of (X- μ_j) is N*d. The parallel scale is N*d, where d is the dimension of data. Parallel scale means the number of operations that can be executed at same time.
2) Calculate the inverse matrix of covariance matrix, the size of covariance matrix is d * d. For the low dimension data, we can calculate the inverse matrix manually. The parallel scale is d*d. For the high dimension data, it is very difficult to parallel the calculation process.
3) Calculate the matrix multiplication. The multiplication process can be divided into two parts: first, multiply the elements in the corresponding positions in these two matrix; second, summary these products. The elements multiplication process and be fully paralleled. But the performance is limited by summation process.
4) Calculate the determinant of the covariance matrix. The covariance matrix is d * d. Same as the calculating inverse matrix, it is easier to have high parallel scale for the low dimensional data, because the covariance will also have low dimension. But for the high dimensional data, we need to use recursion. The approximate speedup compared with serial is about d.
5) Calculate g_j based on the equation.

6) Similarly, the process of update w_j is: do multiplication first, which can be fully parallel. Then do summation reduction.

In the maximization step, all expressions are listed below. Prior probability of every cluster ϕ_j will be computed. It is the average of every column in weighted probability matrix W. Then the new mean μ_j coordinate will also be calculated. According to the algorithm, mean for every cluster is weighted point values in it. And new covariance matrix $\sum j$ for every cluster will be updated by the data, weighted probability and updated mean.

In order to avoid repeating computation, we will first compute the summation of weighted probability in every cluster. After storing that result, we then compute the mean of the summation which is the prior probability of every cluster.

$$\begin{aligned}
\phi_j &:= \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)}, \\
\mu_j &:= \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}}, \\
\Sigma_j &:= \frac{\sum_{i=1}^{m} w_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} w_j^{(i)}}
\end{aligned}$$

Parallelism steps:
(1). Do the transpose of weight probability W in parallel. (Size of W is N*K) Matrix W transpose is generated.
(2). Do summation and calculate average of every row in W transpose in parallel. Then ϕ_j for every cluster has been computed.
(3). Calculate matrix multiplication of W transpose and data. Size of W transpose and data are K*N and N*d. Therefore, the result matrix's size is K*d. Every kernel work item would compute one element in the result matrix.
(4). μ_j for every cluster would be computed by dividing the matrix results with summation of every column in matrix W.
(5). In this step, the updated $\sum j$ would be computed. The first operation is to subtract the cluster μ_j mean from every data point. This could be fully parallelized. After that we could compute the numerator in $\sum j$'s expression. In OpenCL code, $W_j(X- \mu_j) (X- \mu_j)^T$ will be firstly computed in an independent kernel. After this is computed, the summation is computed in parallel. Since the denominator has been computed in step (2), we could easily compute the updated covariance matrix$\sum j$.

V. Performance Analysis:

(1). Local memory vs Global memory
During our implementation, where data should be allocated and where the intermediate computational results should be stored need to be considered. We have tested our program using local memory and that using global memory. The results are listed in table1. We tested K-means algorithm program with 100 data points, 9 dimensions and 4 clusters.

Table1: GPU performance with different memory allocation

CPU execution time	GPU execution time (local memory)	GPU execution time (global memory)
9ms	3.186ms	3.486ms

As listed in table1, local memory behaves better as expected. However as the number of data points increases, memory for data allocation will be increased proportionally. For performance purpose, we use local memory when number of data points is small, local memory would be chosen. Otherwise, we use global memory for storing all data points and intermediate computation results.

(2). Performance Comparison

Table2: Performance of three implementations for K-means

# of points	# of clusters	# of dimensions	CPU time (ms)	GPU time (ms)	GPU time (ms) (Approximation)	Cut-off iterations
100	4	9	9	3.486	1.743	3/6
4000	4	2	389	174.276	87.138	6/12
40000	4	2	4562	2594.892	1297.446	6/12
40000	4	16	40081	20008.490	2589.334	74/85

In the table2 listed below, we compare performance of three implementations for K-means. General speedup of GPU implementation is about 2x. And we also compare the performance of GPU implementation and performance of approximate GPU implementation. Cut-off iterations here mean how many iterations could be removed for an estimated result.

As we can see from table2, speedup in GPU did not increase as number of points increases. Here are some possible reasons. First is about the summation, parallelizing reduction operation like summation here doesn't guarantee a linear speedup. In our implementation, the speedup of the summation itself would be about $N/\log_2 N$. Secondly, there is a matrix multiplication in the implementation. As we discuss, every kernel executes N elements' multiplications and summation of N elements. Therefore, there are only K*d kernels work in parallel and the speed up is far less than N. Additionally, to put every operations accomplished in GPU side, extra matrices are generated as intermediate computational results, which is the overhead compare to CPU implementation. Meanwhile, every memory access is between global memory and GPU processor makes a longer latency than expectation. Some barrier functions also are used for data synchronization. In a short conclusion, the speedup of GPU is slowed than number of points.

Here is the discussion about how to do the approximation. In our experiment, we tested 2 kinds of criterion. The first is that the iterations would be stopped only when differences between the previous center and updated center is smaller than some thresholds, while in the other iteration, how many nodes change their clusters has been counted to decide whether the next iteration would be continued. Table 3 below shows the results. As we can see from the table, all 4 data sets only need about 5

iterations so that the cluster change in nodes will reduce to 1% of the whole data set. As to the center modification difference, the first three data sets would have about 50% cut-off while the last one cut about 7/8 iterations. It shows that only after a very few iterations, K-means algorithm could get a quite accurate clustering result which saving a lot of iterations than the accurate one.

Table3: Performance of two implementations for EM algorithm

# of points	# of clusters	# of dimensions	Cut-off iterations (Criteria 1)	Cut-off iterations (Criteria 2:1%)	Cut-off iterations (Criteria 2:5%)
100	4	9	3/6	1/6	3/6
4000	4	2	6/12	7/12	8/12
40000	4	2	6/12	7/12	9/12
40000	4	16	74/85	81/85	82/85

Figure5 and figure 6 listed below shows difference between the accurate clustering results and approximate result with first criteria. As you can see in the figures, these two look almost the same except that there is slightly difference between the correct one and approximate one.

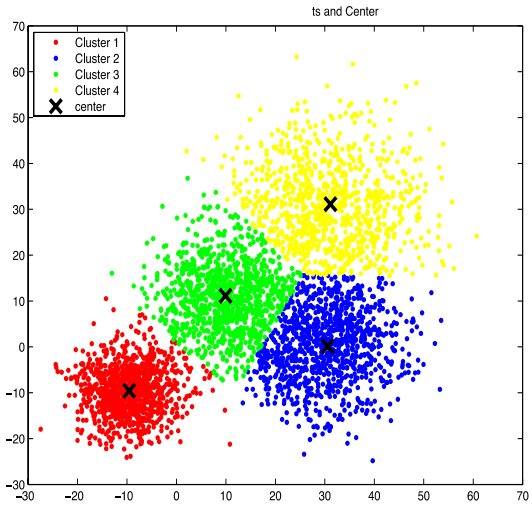

Fig5: Accurate clustering result

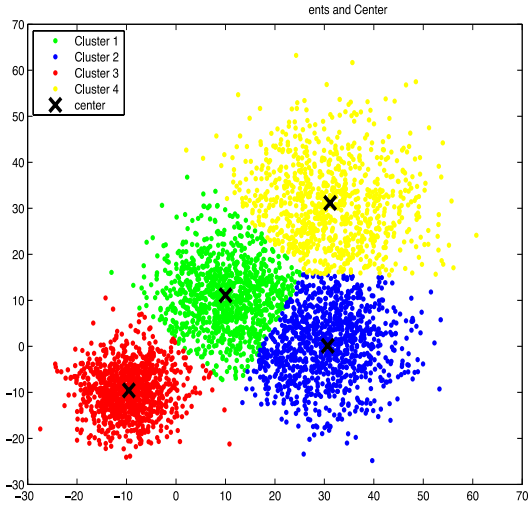

Fig6: Approximate clustering result

In the table4, it shows performance of GPU implementation of EM algorithm. Both the GPU implementation and approximate GPU implementation are listed. More than half of the iterations could be cut off for approximate implementation. As for the speedup, EM algorithm has similar issues with K-means algorithm. Apart from that, usage of double data type and barrier functions slows down the performance as well. And some operations such as matrix inversion and determinant are hard to parallel.

In the figure 7 and figure 8 below, it shows the correct clustering result and approximate result. Similar to K-means comparison result, there are almost the same.

Table4: Performance of different data set with different criterion

# of points	# of clusters	# of dimensions	GPU time (ms)	GPU time (ms) (Approximation)	Cut-off iterations
1000	3	2	3.486	1.743	96/169
4000	4	2	174.276	87.138	125/144
40000	2	2	2594.892	1297.446	157/265

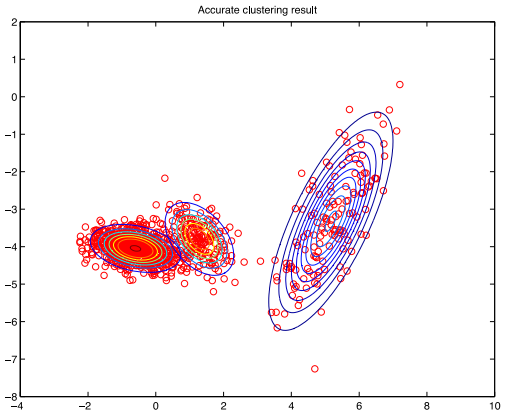

Fig7: Accurate clustering result

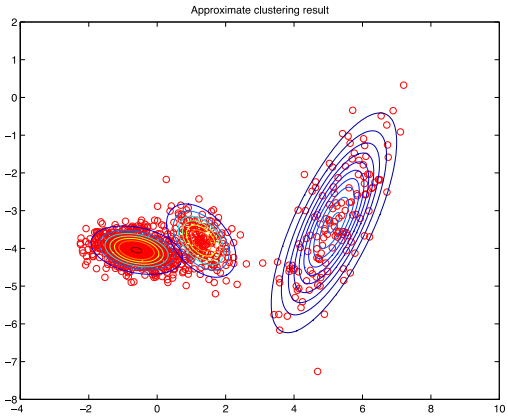

Fig8: Approximate clustering result

VI. Problem Analysis:

Besides the speedups achieved from our proposed approximation-based approach, we also expect to have significant speedups by using GPU. However, the speedups of achieving the accurate results are not as good as we expected. Here we analyze the problems that lower down the speedup:

A. Parallelism for Matrix Multiplication

$$\begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{MN} & \cdots & a_{MN} \end{bmatrix} \times \begin{bmatrix} b_{11} & \cdots & a_{1K} \\ \vdots & \ddots & \vdots \\ b_{N1} & \cdots & b_{NK} \end{bmatrix}$$

Flatten the multiplication process: (1). Compute all $a_{ij}*b_{jk}$ and get M*N*K results (2). Summation for every N results in (1) in parallel.

When doing summation, running the same kernel and same data multiple times gives different results even adding barrier function for synchronization. The reason behind this could be that: Too many operations inside one kernel. The OpenCL compiler optimizes the global ID, so there may cause some mistakes when accessing the elements in array.

Another way to do matrix multiplication is serial and parallel hybrid. Use one row in first matrix and on column in second matrix and do vector multiplication. For each of this operation, it is serial. For the whole matrix, the parallel scale is M*K.

Usually, when people test the performance of matrix multiplication, they use the square matrix. However, in our case, the speedup is limited by the dimension of the dataset and the number of clusters. We have tried one famous linear algebra library clBLAS. Because library API contains many other functionality, for the simple matrix multiplication, its speedup is less than our straightforward approach.
The reasons why we don't choose clBLAS library are:

a. Basic Linear Algebra Subprograms supported by AMD

b. Bad portability, difficult to use on NVIDIA GPU

c. Poor performance (even slower than our implementation)

B. Data race conditions during kernel execution

a. **for** loop in kernel code easily causes data race problems even with barrier. To avoid such condition, either do it serially or call kernel multiple times. Moreover, Barriers necessary in some places. Both two approaches will directly affect the performance.

b. The unbalanced usage of global id. The global work-item ID specifies the work-item ID based on the number of global work-items specified to execute the kernel. If in some parts of the code, a portion of global id is used, the rest unused global id may lead to some errors in the following usage. One way to solve such problem is to partition the big kernel into multiple kernels. However, calling the kernel will also bring the extra overhead.

C. Other Matrix operations

a. Matrix Determinant: As size of matrix increases, recursive methods needed for computation which is not easy for OpenCL implementation

b. Matrix Inversion: As size of matrix increases, it is hard to get solution of the inversion

D. No cache issues

a. For large dataset, we have to define the global variable, which is stored in the global memory of GPU. Every time when the program calls a kernel, it must pass the values of all variables to the kernel. Fetching data when calling many kernels and iterations is very expensive. However, for CPU, frequently fetching data brings very high cache hit ratio. If the size of dataset is not much larger than the cache size, majority of the data can be stored in cache, leading to a very fast data fetching. Moreover, compared with GPU, simple calculations in CPU are not that much time consuming. So if the parallel scale of GPU is not significant, we cannot get significant speedup. Usually the speedup is much lower than the parallel scale.

E. Use GPU in Deepthought2

a. We can compile OpenCL Host device code successfully. But when sbatch jobs with specifying GPU request, it returns segmentation fault.

VII. Conclusion

In this project, we first implemented two most well-known and popular machine learning algorithms in GPU. Second, we proposed an approximation framework and introduced two approximation criteria. From the experimental results, we conclude that by taking advantage of tolerance of inaccuracy, we can efficiently reduce the workload and increase the speedup. In addition, our approach can also increase the energy efficiency of the system.

IIX. References

- Khronos OpenCL Working Group. "The OpenCL Specification, version 1.0. 29, 8 December 2008." *U RL http://khronos. org/registry/cl/specs/opencl-1.0* 29.
- CUDATM, N. "OpenCL Programming Guide for the CUDA Architecture."*NVIDIA Corporation* (2009).
- Zhang, Qian, et al. "ApproxIt: An approximate computing framework for iterative methods." *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 2014.
- http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf
- http://sa09.idav.ucdavis.edu/docs/SA09_NVIDIA_IHV_talk.pdf
- https://chrisjmccormick.wordpress.com/2014/08/04/gaussian-mixture-models-tutorial-and-matlab-code/