Progressive Clustering of Big Data with GPU Acceleration and Visualization

Jun Wang[1], Eric Papenhausen[1], Bing Wang[1], Sungsoo Ha[1], Alla Zelenyuk[2], and Klaus Mueller[1]

[1]Visual Analytics and Imaging Lab, Computer Science Department
Stony Brook University, Stony Brook, NY, USA
Email: {junwang2, epapenhausen, wang12, sunha, mueller}@cs.stonybrook.edu

[2]Chemical and Material Sciences Division, Pacific Northwest National Laboratory, Richland, WA, USA
Email: alla.zelenyuk@pnnl.gov

***Abstract*—Clustering has become an unavoidable step in big data analysis. It may be used to arrange data into a compact format, making operations on big data manageable. However, clustering of big data requires not only the capability of handling data with large volume and high dimensionality, but also the ability to process streaming data, all of which are less developed in most current algorithms. Furthermore, big data processing is seldom interactive, which stands at conflict with users who seek answers immediately. The best one can do is to process incrementally, such that partial and, hopefully, accurate results can be available relatively quickly and are then progressively refined over time. We propose a clustering framework which uses Multi-Dimensional Scaling for layout and GPU acceleration to accomplish these goals. Our domain application is the clustering of mass spectral data of individual aerosol particles with 8 million data points of 450 dimensions each.**

Keywords—clustering, big data, GPU, visualization

I. INTRODUCTION

Big data is everywhere we turn today, recording and affecting everyone and everything– the internet of things, social networks, economy, politics, astronomy, health science, film industry, military surveillance – just to name a few. The vast development of modern technology has made data never so easy to acquire for humankind. And this is especially meaningful for scientific research which has been revolutionized by big data in the past decade. For instance, Nature and Science have even published special issues dedicated to discussing the opportunities and challenges brought by big data [1][2].

While it is generally believed that there can be huge academic and business opportunities emerging from big data, with the fast-growing volume and variety, the task of automated or semi-automated managing and extracting useful knowledge becomes more complex than ever. A major challenge, among all, is that the big storage required by the unprecedented growth of data volume has made the analytical, process, and retrieval operations of big data very difficult. A plausible solution is to label the data points so that they can be arranged into a compact format, for instance, via hierarchical indexing [3] and stratified sampling [4], making operations of big data more efficient and manageable regarding both time and memory cost.

Although crowdsourcing, for instance, via Amazon Mechanical Turk [5], can be leveraged in some smaller-scale cases to obtain data labelling, ultimately, it is impossible for human to match the fast-growing of the big data volume. As a result, automated clustering is often considered a more plausible way and thus an unavoidable first step in big data processing. However, current clustering techniques will mostly fail in the big data context due to their incapability of handling data of large volume and high dimensionality [6], as well as handling data streaming. All this makes scalable incremental clustering of large-scale data so far an unsolved challenge.

In this paper, we tackle this challenge by proposing a novel scalable algorithm which parallelizes *incremental k-means clustering* [7]. The CPU version of the algorithm runs in an incremental way such that data points are read sequentially and each new point is compared to all existing clusters to see if it belongs to a certain cluster or should be recognized as a new cluster. However, such an algorithm can become extremely compute-intensive for big data since the points coming at a later time will have to be compared against all of the cluster centers that have come before it. In contrast, our parallelized algorithm iteratively reads unclustered points in a dataset and parallelly builds clusters in batches on the GPU. We will also suggest proper distance thresholds to users and apply dimension reduction to further boost the performance, both of which are GPU-accelerated.

Our approach is necessitated by our domain application – the analysis of large-scale datasets acquired from a single particle mass spectrometer. We have implemented and tested our algorithm on a single-GPU platform as well as with multiple GPUs. However, the response will not be immediate on either. The best possible compromise is to give the user a glimpse of the partial result that can convey a good hint on what to expect when all is done. To communicate these evolving results, we have opted for visualization that uses Multi-Dimensional Scaling (MDS) to generate a dynamic 2D display of the emerging clustering results. Visual hints are given allowing users to appreciate relevance, updates and changes to the evolving landscape.

Our paper is structured as follows. Section 2 discusses related work. Section 3 presents relevant background. Sections

4 and 5 describe our framework. Section 6 presents results, and Section 7 ends with conclusions and future work.

II. Related Work

A broad survey of clustering algorithms for big data has recently been given by Fahad et al. [6]. In general, these algorithms can be categorized into five classes – partition-based (some well-knowns are *k*-means [8], PAM [9], FCM [10]), hierarchical-based (e.g. BIRCH [11], CURE [12], Chameleon [13]), density-based (DBSCAN [14], OPTICS [15]), grid-based (CLIQUE[16], STING [17]), and model-based (MCLUST [18], EM [19]). However, most of these algorithms cannot handle data on extreme scale, at least under a CPU implementation, due to their low computing efficiency.

To accelerate the computation, parallel algorithms utilizing either distributed architectures or GPUs have been widely studied. One that is most often being adapted is the *k*-means algorithm. Typically, *k*-means has two iterative steps. Step 1 begins with *k* samples (the means) and assigns all other data points to the closest of these *k* means. Step 2 then computes a new mean for each of the k clusters upon which a new iteration begins. Iterations will continue until the total sum of errors falls below some threshold. As *k*-means does not guarantee the global minimum, users usually have to run the algorithm multiple times, also with different numbers of *k*.

A typical approach to parallelize *k*-means on distributed architectures interconnected via MPI/OpenMP is to partition the *N* data points onto *P* processors. Then each processor runs step 1 and 2 on its local data, and the global *k* means are found by averaging the local ones. This can occur in a map-reduce fashion [20][21], in which all mappers distribute their local *k* means to a set of *P* reducers which perform the averaging in parallel. The reducers then send the global *k* means back to the mappers for a new iteration. An alternative approach is to make each processor broadcast its local set of *k* means to all other processors which then all compute the global set locally. This is less parallel but requires less communication, thus better fits the situation where remote workstations are connected via TCP/IP [22].

Finally, the k-means algorithm has also been accelerated on GPUs [9][23][24] using CUDA. Most approaches typically only parallelize step 1 but not step 2 since the number of clusters is usually too low for parallelization. Our approach also uses GPUs and is implemented with CUDA but our purpose is not standard *k*-means where a fixed number of *k* clusters can have any extent as long as they do not overlap with other clusters. Rather, in our method, clusters cannot have an extent greater than a preset threshold, and the number of clusters need not to be specified by the user. This makes a direct comparison to the existing, more general work.

III. Background

As mentioned, the clustering framework presented in this paper is necessitated by our domain application and is actually part of a larger visual analytic system we have been developing in the past ten years [7][25][26][27], collaborating with a group of aerosol scientists. The data is acquired by a state-of-the-art single particle mass spectrometer termed SPLAT II [28], recording 450-dimension mass spectra of individual aerosol particles. SPLAT II can acquire up to 100 particles per second at sizes between 50-3,000 nm with a precision of 1 nm. These data are used to understand the processes that control formation, physicochemical properties and transformations of particles relevant to nanotechnology, catalysis, combustion, atmospheric chemistry, and national security.

The overall goal is to build a hierarchical structure of millions of collected particles based on their composition, which can then be used in subsequent automated classification of new particle acquisitions, either back in the lab or directly in the field. The tools we have developed to create this hierarchy tightly integrate the scientist into this process. Our system provides a variety of interactive controls that allow the scientists to delineate particle clusters directly in the high-dimensional space – a process which we refer to as cluster sculpting. Interactive and intuitive expert-driven tools for this process are strongly needed since the data are extremely complex and fully automated clustering tools do not return satisfactory results.

Fig. 1 shows the interface of a prototype system, called *SpectraMiner*, with a complete particle hierarchy in form of a radial dendrogram [7][25]. Leaf nodes made up of particles are located in the outer ring. These are then merged into higher level nodes based on their distances. A heap sort algorithm merges the currently nearest pair of nodes until reaching the root at which all nodes have been merged.

Since SPLAT II can acquire 100 particles/s, the number of particles gathered in a single run can easily reach 100,000, which takes just 15 minutes. Even 100,000 is a large number of points to compute the classification tree from, and so we have always relied on clustering with a tight bound to detect and remove redundant data points. Since in the onset the number of points was reasonably small, this clustering could be done on the CPU. But now, the experiments and field campaign are much longer and more frequent and so datasets of 5-10M particles have become the norm. This paper uses the dataset acquired during a month-long CARES field campaign in Sacramento, CA [29] in which SPLAT II operated 24/7 for the entire month. To keep the size of the dataset manageable, the sampling rate was reduced to 20 particles/s. The CPU solution was insufficient to perform the clustering at this level of magnitude, which necessitated our high-performance GPU solution.

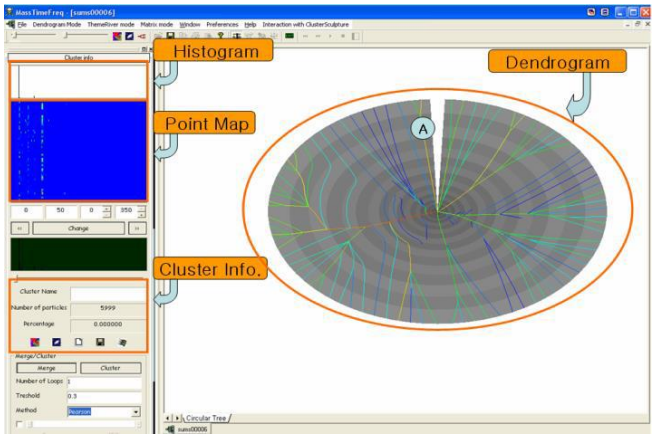

Fig. 1 The SpectraMiner interface

IV. Our Approach

The algorithm we present here adapts and parallelizes the incremental *k*-means algorithm proposed by Imrich et al. [7], making it amenable to GPU acceleration. The incremental clustering algorithm has the desirable property that the value of *k* does not need to be predetermined. However, it was not structured for a parallel implementation.

A. Incremental K-Means

The pseudo code of the incremental *k*-means algorithm is given in Alg. 1. The algorithm starts with making the first point of a dataset the initial cluster center, and then scans through all unclustered points. Each unclustered point *p* is compared to all found cluster centers. A point is clustered into the nearest center *c* if their distance, calculated by the function *distance(p, c)*, is within some predefined threshold *t*, otherwise it is made a new cluster center for later points. The process stops when all points are clustered. The distance threshold *t* acts as the regulator for the clustering result, such that a larger *t* leads to a smaller number of clusters each with more points, while a small *t* could result in many small clusters.

As the algorithm is running, it also keeps track of any small clusters that have not been updated for a while and marks them as outliers. Then, after all points are clustered, a second pass is performed to re-cluster the points in these outlier clusters. Different from that in the first pass in which each point is only compared to the cluster centers coming before it, each outlier point can be clustered into any cluster center generated during the first pass.

One important advantage of the incremental *k*-means over other common clustering algorithms is that it can handle streaming data. Each new data point in a stream can be simply added to the nearest cluster or made into a new cluster center depending on the distance threshold. However, Alg. 1 is not very scalable and can gradually become slower with growing data size and number of clusters, as the points coming at a later time will have to be compared against all cluster centers that came before it. This can become extremely expensive regarding the time cost in the big data context, especially when the points are of high dimensionality and the cost of calculating the distance function *distance(p, c)* takes a non-negligible amount of time.

B. Parallel Incremental K-Means

The sequential nature of the incremental *k*-means makes it unclear how to map it to GPUs efficiently. The most naïve way would be to parallelize over the cluster centers such that each unclustered point is compared to all centers in parallel. However, there are a few problems with this approach. First, the GPU would be highly underutilized at the beginning period of the algorithm when there are too few clusters. Also, the points would still be iterated through in a sequential fashion which is actually the costliest part of the algorithm. Thus, a better choice could be to parallelize over the points so that all unclustered points are processed in parallel, comparing them to each cluster. Nevertheless, this approach has the practical difficulty that a large dynamic memory must be maintained on the GPU to handle the increasing number of clusters, along with the running of the algorithm. This is impossible, especially when processing data of large volume with potentially numerous clusters, since that the number of clusters cannot be known in advance, and that the GPU memory is typically very limited and must be pre-allocated in fixed size before each invoking.

Algorithm 1: Incremental K-Means

```
Input: data points P, distance threshold t
Output: clusters C
C = empty set
for each unclustered point p in P
    if C is empty then
            Make p a new cluster center and add it into C
    else
            p = next unclustered point
            Find the cluster center c in C closest to p
            let d = distance(c, p)
            if d < t then Cluster p into c
            else Make p a new cluster center added to C
            end if
        end if
end for
return C
```

Our solution, in consideration of all these issues, is to build clusters in batches, so that unclustered points and cluster centers in the batch can be parallelized over at the same time. The parallelized algorithm runs iteratively over unclustered points and builds clusters incrementally. In each iteration, the algorithm first scans unclustered points sequentially on the CPU to detect a batch of *b* cluster centers, denoted as ***B***. As *b* is typically a very small number, only a few points will be scanned. Then the algorithm parallelly computes the distances between each unclustered point to each center in ***B*** on the GPU. The nearest center for each point is found at the same time so that a point can be assigned with a label of its nearest cluster if their distance is within the predefined threshold. However, a point can be officially assigned to a cluster only on the CPU after the labels are passed back from GPU memory. After this, the centers in ***B*** are updated and the batch is added to the output set ***C*** if it is stable, otherwise the process will be operated again until ***B*** is

Algorithm 2: Parallel Incremental K-Means

```
Input: data points P, distance threshold t, batch size b,
       max iteration M
Output: clusters C
C = empty set
while number of un-clustered points in P > 0
    Run Alg. 1 until a number of b clusters B emerge
    Iteration i = 0
    while i < M and B is not stable
        in parallel:
            for each unclustered point p_i
                Find the center b_i in B closest to p_i
                if distance(b_i, p_i) < t then c_i = b_i
                else c_i = null
            end for
        on CPU: Assign p_i to b_i if c_i is not null
        in parallel: update centers of B
    end while
    Add B to C
end while
return C
```

stable or reaches the max iteration *M*. Empirically, we found *M =5* is effective, but note that most iterations converge much earlier. Clustered points will not be scanned again in later iterations. At last, the outer iteration stops when all points are clustered. Alg. 2 presents the pseudo code of the whole process.

Each outer iteration of Alg. 2 essentially merges Alg. 1 with a parallel implementation of the traditional *k*-means where $k = b$. The advantage here is that a point will never be compared to more than *b* cluster centers. Also, a second pass for re-clustering "outliers" is no longer needed. The batch size *b* controls the workload balance between CPU and GPU, and typically *b* should be a small number. Through experimentation, we chose $b = 96$. We found that setting $b > 96$, although this means less iteration steps, would make the algorithm CPU bound, which means the GPU may have more idle time waiting for the CPU to build the batch. And conversely, a smaller *b* could result in GPU underutilization. The value of *b* is also suggested to be a multiple of 32 to avoid divergent warps with an NVIDIA GPU under CUDA implementation.

C. Determine the Distance Threshold

In both Alg. 1 and 2, the distance threshold *t* plays the central role for adjusting the clustering results. A good value of *t* reflecting the nature of the data can be chosen regarding the distribution of the point distances, e.g. the value indicating the intra-cluster distance of clusters. However, computing the distance matrix could be exhausting especially given a dataset of high dimensionality.

We took two approaches to ease this issue. First, we reduce the number of dimensions by removing the irrelevant ones based on the dimension standard deviations computed from all points. This could also heavily reduce the computational load of the clustering algorithm. Second, when there are too many points, we only use a random sampling of, say, 20,000 to 50,000 points instead of all of them. The process of computing the distance histogram as well as computing the dimension standard deviations are all GPU accelerated to further boost the speed. Details of the implementation are given in section 5.

D. Cluster Visualization

We aim for a progressive display that can intuitively show the evolving data clusters along with the running of the clustering algorithm, so that users can estimate the final result at an early stage rather than waiting for the finish. The MDS is such a visualization for this purpose providing a low-dimensional embedding of the data into a 2D plane. Here we leverage Glimmer MDS [30] which is an iterative algorithm where points are embedded (i.e. reduced to a 2D layout) in the current level based on the embedding of the points at a previous level. The main advantage of Glimmer MDS is that the final embedding of a point is only decided by a small number of points. Although the set of referential points change at every iteration, the size remains the same and thus the computation cost is fixed, making the algorithm amenable for GPU acceleration.

However, for our data (or any large-scale data), even a part of it can contain too many points such that the MDS visualization can easily become very clustered if we visualize all of them. Thus, a better approach is to only visualize the cluster center which is a tight representative of its members. Further, we also provide users options to filter clusters such that only *significant* clusters with more than a certain number of members are displayed. We also visualize the size distribution of clusters as histograms.

The color of a point in the MDS visualization then represents the number of members in the cluster. We use a color map from white to blue, in which small clusters are mapped to mostly white and large clusters to saturated blue. As the background color is white, those significant clusters are emphasized in such mapping. However, the range of the number of members in each cluster can vary widely (e.g. from 1 to 317,786 in one of our applications), and the cluster sizes may not be evenly distributed across the range. A linear color mapping function might not effectively catch the difference in such case. Instead, we apply a piecewise transfer function linking the number of members in each cluster to the color saturation change of its representative point in the visualization. By such, we can exercise more control over how points are colored. Then, by incorporating this visualization component into our parallel incremental clustering algorithm we can provide a streaming experience where we initially visualize the first few cluster centers, and then update the view with new cluster centers as they are formed.

V. Low Level Implementation

Our parallel clustering algorithm described in the previous section was implemented on NVIDIA GPUs using CUDA. We took the Pearson distance metric in our application, which can be calculated as $d_{xy} = 1 - \rho_{xy}$ where ρ_{xy} is the Pearson correlation coefficient of two points. We now introduce the detailed implementation in the following.

A. Kernel for Parallel Clustering

For our parallel incremental clustering, each GPU thread block is set to have a thread dimension of 32 by 32. This means each block will compare 32 points to a batch of *b* (which is a multiple of 32) cluster centers. Thus, we lunch *N*/32 thread blocks if we have totally *N* data points. Fig. 2 briefly illustrates the GPU thread block access pattern. The *x* coordinate of a thread tells which point it will be operating on, and the *y* coordinate is mapped to a small group of *b*/32 cluster centers. More specifically, as we set $b = 96$, each thread will process three cluster centers ($96/32 = 3$). The cluster centers and points are stored in GPU memory as two matrices with the same x dimension. Then, each thread will compute the distances between the corresponding point and the small group of cluster centers, and store the nearest cluster index and its distance value in the shared memory for further processing.

Fig. 3 gives the pseudo code for the GPU kernel operated by each thread, where the two 32 by 32 shared memories are denoted *distance[][]* and *cluster[][]*. Each row of the *distance[][]* stores the distances between a point and its nearest cluster center in the small centers group. That is to say, each element of *distance[][]* stores the distance of the nearest center among the three that are compared against in a thread. Meanwhile, the index of the corresponding clusters are saved in *cluster[][]*. Then, after synchronizing all the threads in the block so that all shared memories are filled with stable results, each

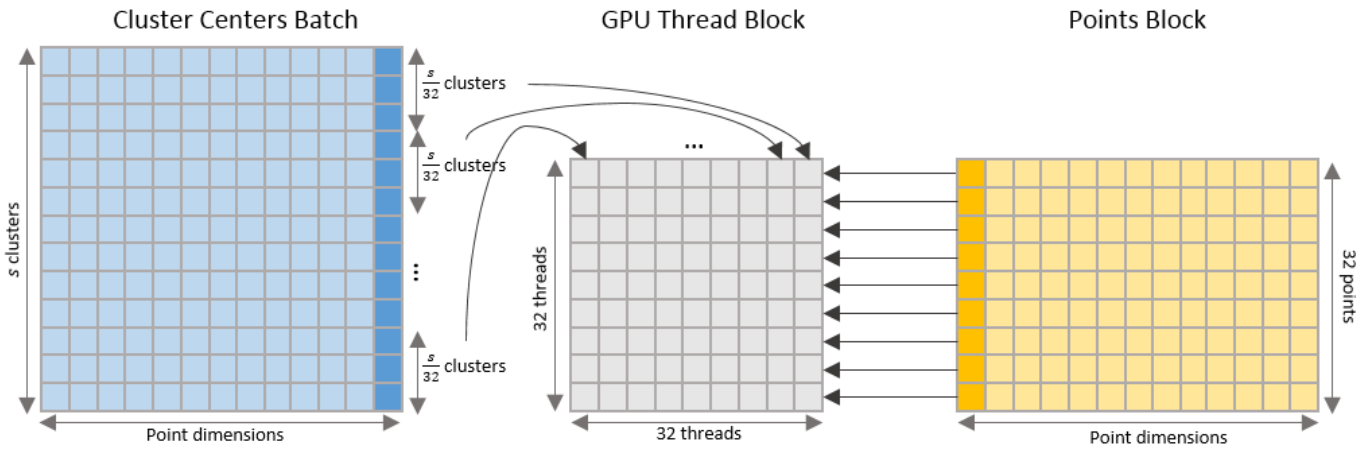

Fig. 2. GPU access pattern for the parallel clustering algorithm

```
pid = blockDim.y * blockIdx.x + tid.x          // point id
C = centers to compare

distance[tid.y][tid.x]  =  minimum  distance  between
        point[pid] and centers in C
cluster[tid.y][tid.x] = id of the nearest cluster center in C
syncthreads()

if tid.x equals 0 then
        min_dis = minimum of the row distance[tid.y]
        if min_dis < distance threshold then
                label[pid] = the corresponding cluster id
        sored in the row cluster[tid.y]
        else label[pid] = -1
        end if
end if
```

Fig. 3. Pseudo code of the CUDA kernel for parallel clustering

thread with x id of 0 will scan through one row of the shared memory, looking for the minimum distance and the nearest cluster. A point will be labeled with the nearest cluster id if the calculated distance is within threshold, otherwise -1 indicating the point is not clustered in the current iteration of the algorithm (see Alg. 2). After all thread blocks finish their job, the labels of all the points will be returned and be used for CPU to officially assign points to clusters.

As mentioned, the batch size b can directly influence the per-thread workload. A larger b means each thread will have to compare more cluster centers. The workload of GPU kernels is also affected by the dimensionality of data and the computing complexity of the distance metric. For our typical application, we found that setting $b = 96$ could reach the best load balance between GPU and CPU, although the choice may vary for other datasets and different computing platforms.

B. Standard Deviations and Pairwise Distances

As mentioned, we perform simple dimension reduction to accelerate the computation. This is achieved by removing dimensions with small standard deviations, e.g. by a threshold of 0.01 of the max value of all standard deviations. The computation of standard deviations is done on GPU with an optimization technique called *parallel reduction* [31]. The technique takes a tree-like iterative approach within each GPU thread block, summing up all values mapped to each thread the block (illustrated in Fig. 4).

As the data of one dimension forms a very long vector, the calculation of the mean as well as the standard deviation of the vector can be transferred into a vector reduction operation. For our CUDA implementation, dimensions are mapped to the y-coordinates of thread blocks. We launch 512 threads in a block, each mapped to the value of one data point in one dimension, i.e. each block has a thread dimension of 512×1. Thus, the block dimension is $D \times N/512$, where N is the number of points and D is the data dimensionality. Each value mapped to a thread is initialized in the beginning, depending on the goal of the function. That is to say, for calculating the vector mean μ, each value is initially divided by N, and for calculating the vector variance, each value x_i is mapped to $(x_i - \mu)^2/N$. And then in each iteration step, the number of active threads in a block is halved, and the values of the second half of the shared memory are added to the first half, until there is only one active thread getting the final result of the block and storing it into the output vector. The pseudo code of the GPU kernel is given in Fig. 5.

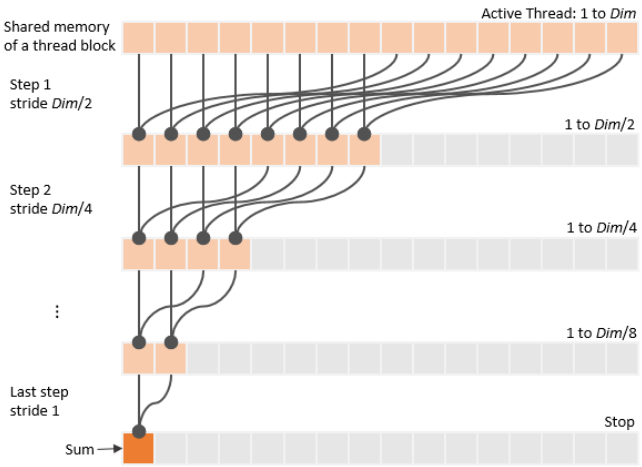

Fig. 4. GPU thread block iterations of Parallel Reduction

```
pid = threadIdx.x // in thread point id
    d = blockIdx.y // dimension id
        i = blockIdx.x * blockDim.x + pid // point id
                    // shared memory
        sdata[tid] = initialized value of point[i][d]
                    s = blockDim.x / 2
                        while s > 0
                          if pid < s then
            sdata[pid] += sdata[pid + s]
    s = s / 2 and syncthreads()
                        end while
          if pid == 0 then output sdata[0] to result
```

Fig. 5. Pseudo code of the CUDA kernel for Parallel Reduction

The result of the parallel reduction operation is a vector down-scaled by 512 times of the input. If this is still too large, we can conduct it again until we reach we reach the final output of a single summed value. However, as the cost of memory transfers may be higher than the benefit we can get from GPU parallelization when processing a short vector, a single CPU scan would be more than sufficient in such case.

One practical difficulty, especially when implementing on a single GPU, is that there may not be enough GPU memory that can hold all the data. Even if there is, the length of the data array can go beyond the maximum indexable value such that they cannot be accessed. Our solution is to divide data into blocks of the size that can be held in GPU memory, and operate parallel reduction on each of them. Then an extra CPU scan is operated on results from data blocks to summarize the final output.

The computation of the pairwise distance histogram faces a similar problem. Although we can sometimes fit the sampled data points in GPU memory, the length of the result vector can easily go beyond the indexable range (e.g. the pairwise distances of 50,000 points can form a vector of 1,249,975,000 elements). Then again, we divide sampled points into blocks of fixed size. As we only need the histogram, we update the statistics on the CPU whenever the distances of points from two blocks are returned by the GPU and then drop the result to save memory. The access pattern of GPU thread blocks for computing pairwise distances is straightforward – each thread calculates one pair of distances. As we use 32×32 thread blocks, there will simply be $N/32 \times N/32$ blocks launched.

VI. Results

We have implemented our parallel incremental clustering algorithm (Alg. 2) on a server with 4 Tesla K20 GPUs. We also implemented the sequential algorithm presented earlier (Alg. 1). We tested our algorithm on the aerosol dataset introduced in Section 3. The total dataset contains 8 million points each in 450 dimensions.

A. Dimension Reduction and Distance Threshold

We first perform the dimension reduction regarding the standard deviations presented in Fig. 6a. Here, by applying a threshold of 0.01 of the maximum value calculated, 36 dimensions are selected out of 450 and marked red in Fig. 6a, while the rest colored blue (most of which are too small to be observed clearly).

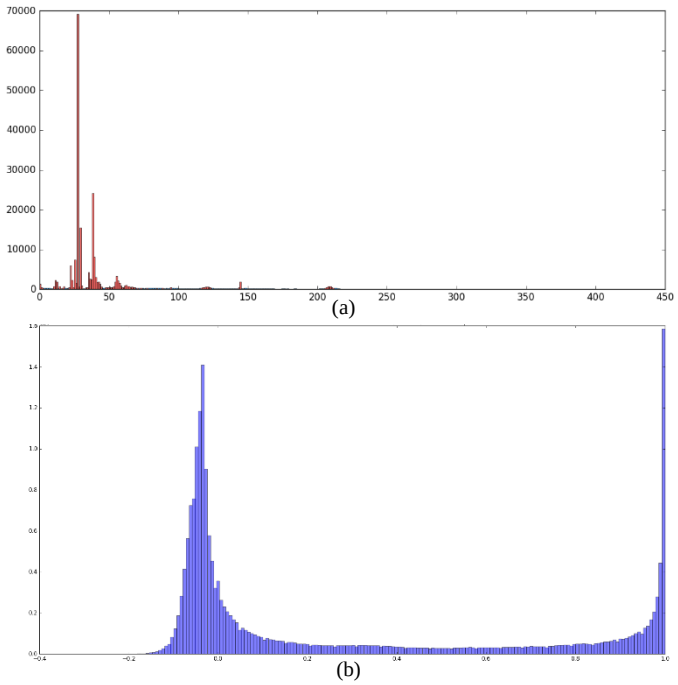

Fig. 6. Deciding the distance threshold. (a) Standard deviations of dimensions. The red bars correspond to the selected dimensions. (b) The distribution of pairwise Pearson distances from 20,000 sampled points. A threshold of 0.3 is selected regarding the gap in the middle.

The histogram of pairwise Pearson distances between each two of 20,000 samples is presented in Fig. 6b. We also have calculated such histograms with more samples, e.g. 50,000 random points, but the shape of the histogram is basically the same as that in Fig. 6b, just with enlarged values for all the bars. This means 20,000 samples have been quite enough for describing the distance distribution. We can see there is a clear gap in the center between the peaks at the two ends of the distribution indicating the general intra and inter distances of clusters. Based on Fig. 6b, the user-defined distance threshold was set to a Pearson distance of 0.3 throughout our experiments.

B. Clustering Quality and Timing

The clustering quality is measured using the Davies-Bouldin (DB) index [32], calculated as

$$DB = \frac{1}{n}\sum_{i=1}^{n} \max\left(\frac{\sigma_i + \sigma_j}{M_{ij}}\right) \quad (1)$$

Here, σ_i is the dispersion measure of cluster i calculated as the average distance of all elements in cluster to the center, M_{ij} is the Pearson distance of two centers (dissimilarity measure), and n is the total number of clusters. With the DB index, the lower the score, the higher the quality of the clustering.

In our experiments, we noticed that the sheer size of our datasets was the main performance bottleneck. With millions of points, each call to the GPU would take between 2-4 seconds. Since the GPU was being called thousands of times, this was

still a very time-consuming process. By removing points that were considered "close enough" to their respective cluster centers, the size of the dataset would decrease with every call to the GPU. This optimization, which we call *sub-thresholding* (ST), drastically reduced the computation time. By setting the sub-threshold to 0.2, for example, any point that has a Pearson distance of less than 0.2 for their current cluster will become ineligible for re-clustering (i.e. the point will stay in that cluster even if a closer cluster is introduced later). This effectively prevents points with a low intra-cluster distance from moving to a new cluster in a future iteration. As a result, these points can be removed from consideration.

Table 1 compares the DB index of our parallel clustering algorithm to the sequential algorithm. Table 1 also shows how different ST levels affect quality. Unexpectedly, clustering with no ST does not produce the best results. This suggests that there is some discrepancy between what is locally optimal (i.e. each point lies in cluster where it is closest to the cluster center) and what is globally optimal (i.e. producing clusters with a low intra-cluster distance and a high inter-cluster distance).

TABLE I. DB SCORES UNDER DIFFERENT SETTINGS

Size	Sequential	Parallel	ST: 0.3	ST: 0.2	ST: 0.15
10k	**0.527**	0.539	0.540	0.537	0.529
50k	0.546	0.590	0.548	0.554	**0.539**
100k	0.550	0.584	0.600	0.570	**0.544**
200k	**0.564**	0.587	0.640	0.593	**0.564**

DB scores measuring the cluster quality (lower is better). Quality of sub-thresholding (ST) is also presented.

Intuitively, this makes sense. Because the points are of high dimensionality, it is likely that if *m* points find clusters that are closer, they will be in *m* separate clusters. Within each cluster, a single point will have a small impact on the intra-cluster distance. The cluster that is losing *m* points, however, is going to suffer a large increase in its intra-cluster distance if those points were relatively close to the center. In addition, the *m* updated clusters will have a smaller inter-cluster distance.

Fig. 7 shows how the timings scale with increasing data volume under different clustering settings. The graph compares the sequential to the parallel clustering with differing values of ST. Here we can see that our parallel approach is significantly faster than the sequential one. What's more, as the data volume increases, the sub-thresholding optimization effectively helps to suppress the explosion in computing time.

Also, we can see from Fig. 7 that the computing time, under any setting, grows at a super-linear rate with respect to the size of the data (i.e. more than double the compute time is required for processing doubled data). This makes the algorithm a good candidate for multi-GPU acceleration. Therefore, we have also implemented such a multi-GPU solution where the same cluster batch is passed to all CPUs and unclustered points are distributed to all GPUs evenly. The only difference from the single-GPU implementation is that one extra step is added on the CPU to merge the centers after all GPUs have finished clustering.

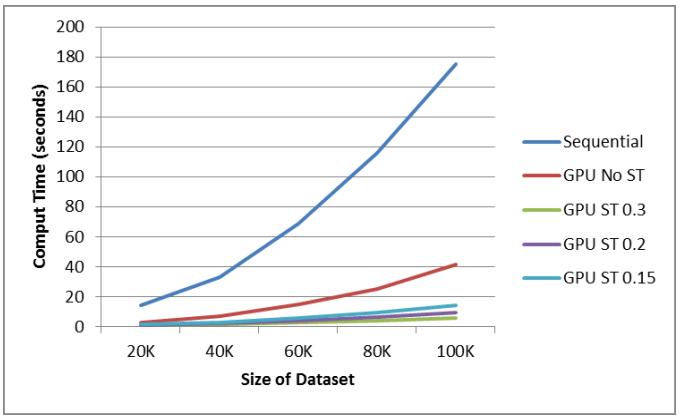

Fig. 7. Timings under different clustering with increasing data sizes.

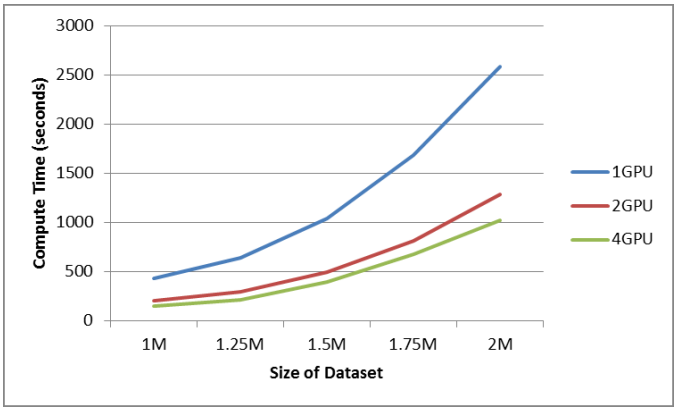

Fig. 8. Timings of the multi-GPU implementations (ST: 0.15)

Fig. 8 shows the timings under different numbers of GPUs scaling with data sizes. The ST is set to be 0.15 as it performs overall best regrading Table 1. We can see here the time needed is basically halved when using two GPUs comparing to using a single GPU. This is because each GPU is only working on half of the dataset. However, such a speed-up is not observed when increasing the number of GPUs from two to four. This suggests that there is a diminishing rate of return in the multi-GPU approach, probably caused by the increasing cost in data transferring and more needed work in merging the results from different GPUs.

C. *Visualizations*

Fig. 9 shows some visualizations generated during the incremental clustering. For each state of the clustering result, we visualize the MDS layouts of cluster centers (on the left) as well as the histogram of the size of all clusters (on the right). The number of significant cluster centers being visualized (i.e. with more than 10 members) out of the total number of clusters processed is displayed on the top-left corner of each MDS layout. It is worth noting that the clusters appear rotated in MDS from plot to plot. This is purely due to the Glimmer MDS that does not preserve certain orientations.

From the visualizations, we notice two interesting facts. First, several clusters are forming in the MDS layout. We believe this is due to the low threshold enforced by the

incremental clustering algorithm. Some clusters are actually close but not enough to form one single cluster. Second, the ratio of the significant clusters over the total clusters being processed is getting smaller. In the beginning the ratio is 82.3% but toward the end is only about 1%. From the last two figures, we also see that of the first 165,984 clusters, 4,002 of these are significant (>10 points), while for the second 171,010 clusters only about 200 are significant. This is because most of the big clusters were generated during the early rounds and most of the clusters produced in the late iterations only have a small number of members. This is a major advantage of the visual feedback – users can easily determine when partial clustering results are sufficient to perform a subsequent analysis step.

VII. Conclusion and Future Work

We have presented a novel algorithm for big data clustering. The algorithm parallelizes the incremental k-means and is implemented on the GPU with CUDA. We observe in experiments that the new parallel algorithm can speed up the clustering process by an order of magnitude and lead to a smaller growth of time cost along with the increasing of data size.

Future work will include an out of core approach to handle even larger datasets. Although we are able to fit current data in RAM, ultimately, we need to adopt more efficient data storage facilities and load balancing strategies. For example, in the multi-GPU approach, we can redistribute points onto each GPU after every call to the parallel clustering function. Additional experiments will be done to determine if the latency of the extra data transfer is small enough to reach a net gain in performance.

The overall goal of our domain application, as mentioned in Section 3, is to obtain the structural relationship embedded in data which can be used to classify new particles at real-time. The parallel clustering algorithm is the first step towards this goal. Although we have been able to build such structures on smaller datasets with a simple heap sort algorithm, this strategy becomes clumsy when facing big dataset where the number of leaf nodes are usually very large. A possible solution can be to build the structure with parallel clustering in a similar fashion as *k*-means trees [3]. This will be the major focus of our future work.

Acknowledgement

This research was partially supported by NSF grant IIS 1527200 and the Ministry of Science, ICT and Future Planning, Korea, under the "IT Consilience Creative Program (ITCCP)" supervised by NIPA. Partial support (for Alla Zelenyuk and Jun Wang) was also provided by the US Department of Energy (DOE) Office of Science, Office of Basic Energy Sciences, Division of Chemical Sciences, Geosciences, and Biosciences.

References

[1] "Big data," *Nature*, vol. 455, no. 7209. pp. 1–136, 2008.

[2] "Dealing with data," *Science*, vol. 331, no. 6018. pp. 639–806, 2011.

[3] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 11, pp. 2227–2240, 2014.

[4] "Stratified sampling," *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Stratified_sampling. [Accessed: 20-May-

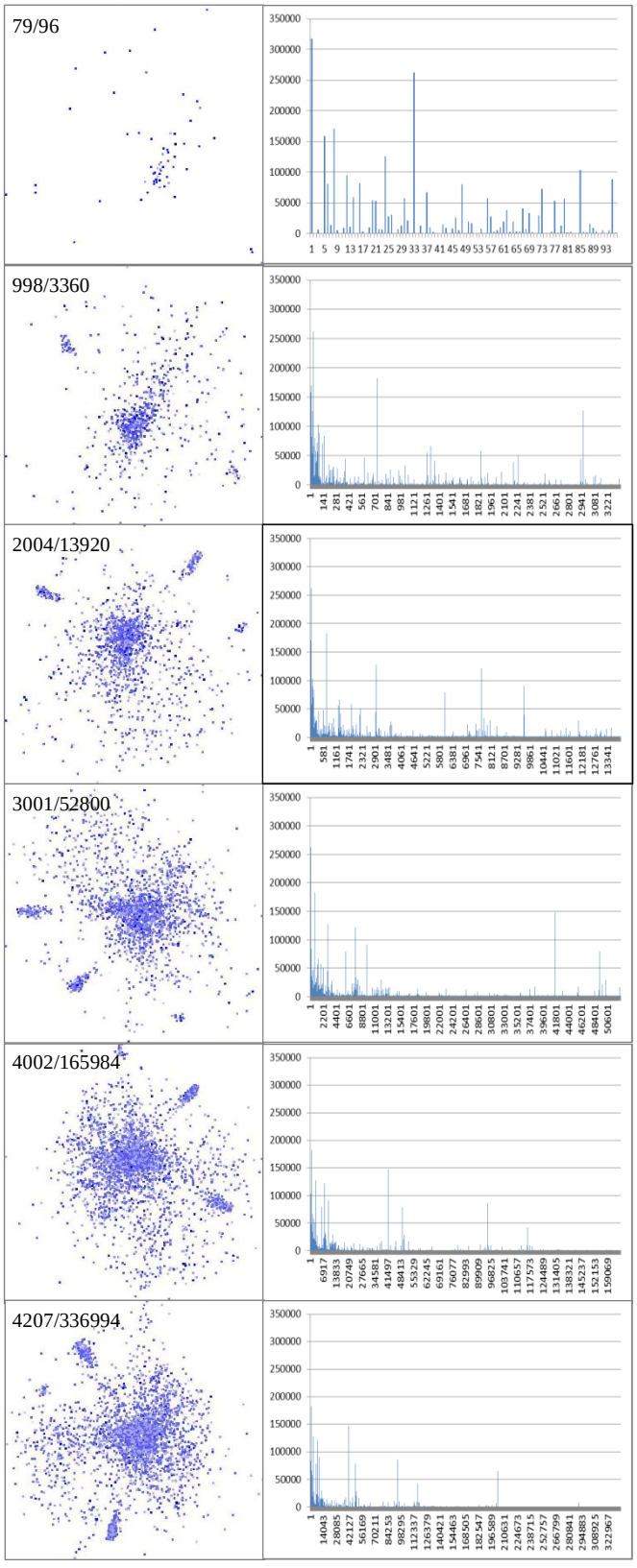

Fig. 9. Cluster visualization along with the clustering process. The number on the top-left corner of each pair of figures is the number of clusters being visualized vs. the total number of clusters. Clusters with less than 10 members are not visualized.

2017].

[5] "Amazon Mechanical Turk." [Online]. Available: https://www.mturk.com/mturk/welcome.

[6] A. Fahad *et al.*, "A survey of clustering algorithms for big data: Taxonomy and empirical analysis," *IEEE Trans. Emerg. Top. Comput.*, vol. 2, no. 3, pp. 267–279, 2014.

[7] P. Imrich, K. Mueller, R. Mugno, D. Imre, A. Zelenyuk, and W. Zhu, "Interactive Poster: Visual data mining with the interactive dendrogram," in *IEEE Information Visualization Symposium*, 2002.

[8] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *the fifth Berkeley symposium on mathematical statistics and probability*, 1967, vol. 1, no. 14, pp. 281–297.

[9] R. Farivar, D. Rebolledo, and E. Chan, "A parallel implementation of k-means clustering on GPUs," in *PDPTA*, 2008, pp. 340–345.

[10] J. C. Bezdek, R. Ehrlich, and W. Full, "FCM: The fuzzy c-means clustering algorithm," *Comput. Geosci.*, vol. 10, no. 2–3, pp. 191–203, 1984.

[11] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases," *ACM SIGMOD Rec.*, vol. 25, no. 2, pp. 103–114, 1996.

[12] S. Guha, R. Rastogi, and K. Shim, "CURE: An efficient clustering algorithm for large databases," *Inf. Syst.*, vol. 26, no. 1, pp. 35–58, 2001.

[13] G. Karypis, Eui-Hong Han, and V. Kumar, "Chameleon: hierarchical clustering using dynamic modeling," *Computer (Long. Beach. Calif).*, vol. 32, no. 8, pp. 68–75, 1999.

[14] M. Ester, H. P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226–231.

[15] M. Ankerst, M. M. Breunig, H. Kriegel, and J. Sander, "OPTICS: Ordering Points To Identify the Clustering Structure," *ACM SIGMOD Rec.*, vol. 28, no. 2, pp. 49–60, 1999.

[16] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic subspace clustering of high dimensional data for data mining applications," *ACM SIGMOD Rec.*, vol. 27, no. 2, pp. 94–105, 1998.

[17] W. Wang, J. Yang, and R. Muntz, "STING: A statistical information grid approach to spatial data mining," in *International Conference on Very Large Data*, 1997, pp. 1–18.

[18] C. Fraley and A. E. Raftery, "MCLUST: Software for Model-Based Cluster Analysis," *J. Classif.*, vol. 16, no. 2, pp. 297–306, 1999.

[19] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *J. R. Stat. Soc. Ser. B*, vol. 39, no. 1, pp. 1–38, 1977.

[20] C.-T. Chu *et al.*, "Map-reduce for machine learning on multicore," in *Advances in neural information processing systems*, 2007, pp. 281–288.

[21] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on mapreduce," in *IEEE International Conference on Cloud Computing*, 2009, pp. 674–679.

[22] S. Datta, C. Giannella, and H. Kargupta, "Approximate distributed k-means clustering over a peer-to-peer network," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 10, pp. 1372–1388, 2009.

[23] B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He, "K-Means on Commodity GPUs with CUDA," in *2009 WRI World Congress on Computer Science and Information Engineering*, 2009, pp. 651–655.

[24] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via CUDA," in *Proceedings of the 1st International Conference on Intensive Applications and Services, INTENSIVE 2009*, 2009, pp. 7–15.

[25] A. Zelenyuk, D. Imre, Y. Cai, K. Mueller, Y. Han, and P. Imrich, "SpectraMiner, an interactive data mining and visualization software for single particle mass spectroscopy: A laboratory test case," *Int. J. Mass Spectrom.*, vol. 258, no. 1–3, pp. 58–73, 2006.

[26] A. Zelenyuk, D. Imre, E. J. Nam, Y. Han, and K. Mueller, "ClusterSculptor: Software for expert-steered classification of single particle mass spectra," *Int. J. Mass Spectrom.*, vol. 275, no. 1–3, pp. 1–10, 2008.

[27] Z. Zhang, K. T. Mcdonnell, E. Zadok, and K. Mueller, "Visual Correlation Analysis of Numerical and Categorical Data on the Correlation Map," *IEEE Trans. Vis. Comput. Graph.*, vol. 21, no. 2, pp. 289–303, 2015.

[28] A. Zelenyuk, J. Yang, E. Choi, and D. Imre, "SPLAT II: An Aircraft Compatible, Ultra-Sensitive, High Precision Instrument for In-Situ Characterization of the Size and Composition of Fine and Ultrafine Particles," *Aerosol Sci. Technol.*, vol. 43, no. 5, pp. 411–424, 2009.

[29] R. A. Zaveri *et al.*, "Overview of the 2010 Carbonaceous Aerosols and Radiative Effects Study (CARES)," *Atmospheric Chemistry and Physics*, vol. 12, no. 16. pp. 7647–7687, 2012.

[30] S. Ingram, T. Munzner, and M. Olano, "Glimmer: Multilevel MDS on the GPU," *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 2, pp. 249–261, 2009.

[31] M. Harris, "Optimizing parallel reduction in CUDA," *NVIDIA CUDA SDK 2*. 2008.

[32] D. L. Davies and D. W. Bouldin, "A Cluster Separation Measure," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-1, no. 2, pp. 224–227, 1979.