This paper was presented at the *First IEEE Workshop on Big Data Visualization*
Santa Clara, CA, October, 2013

# GPU-Accelerated Incremental Correlation Clustering of Large Data with Visual Feedback

Eric Papenhausen[1], Bing Wang[1], Sungsoo Ha[2], Alla Zelenyuk[3], Dan Imre[4], and Klaus Mueller[1,2]

Visual Analytics and Imaging Lab, Center for Visual Computing, Computer Science Department
[1] Stony Brook University, Stony Brook, NY, USA and [2] SUNY Korea, Songdo, Korea
Email: {epapenhausen, wang12, sunha, mueller}@cs.sunysb.edu

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

[4] Imre Consulting, Richland, WA, USA
Email: dimre2b@gmail.com

***Abstract*—Clustering is an important preparation step in big data processing. It may even be used to detect redundant data points as well as outliers. Elimination of redundant data and duplicates can serve as a viable means for data reduction and it can also aid in sampling. Visual feedback is very valuable here to give users confidence in this process. Furthermore, big data preprocessing is seldom interactive, which stands at conflict with users who seek answers immediately. The best one can do is incremental preprocessing in which partial and hopefully quite accurate results become available relatively quickly and are then refined over time. We propose a correlation clustering framework which uses MDS for layout and GPU-acceleration to accomplish these goals. Our domain application is the correlation clustering of atmospheric mass spectrum data with 8 million data points of 450 dimensions each.**

***Keywords-big data; clustering; visualization; visual analytics; correlation; GPU;***

## I. INTRODUCTION

Some have called big data the second major revolution in modern civilization after the industrial revolution [23]. Indeed, big data affects everyone and everything – the internet of things, social networks, internet search indexing, astronomy, atmospheric science, health, biology, military surveillance, e-commerce – just to name a few. Big data is often associated with the terms e-science and data science, which really both refer to a similar goal – the automated or semi-automated extraction of knowledge from massive volumes of data. Just e-science is the term commonly used in astronomy, oceanography, and biology, while data science is the term used in business applications [7].

In data mining and especially in big data, preprocessing consumes a large portion of the workflow, as much as 80-90% [1]. Preprocessing includes data preparation, integration, cleaning, reduction, and transformation. As big data analysis can often be mission critical, preprocessing should be done expediently. The massively parallel architecture of GPUs offers an effective platform to accomplish high speed data preprocessing. However, as GPU technology has been developing steadily and rapidly, users have trouble keeping up. And even if they do, the largeness of big data requires not just one GPU but multiple GPUs in conjunction with large memory. These needs are best addressed in a cloud-based platform. The framework we describe in this paper utilizes both a single GPU as well as a GPU server that could also operate in a cloud setting.

But no matter on which computational platform the preprocessing is conducted, the decimation of big data to a more manageable size brings great benefits. Given the abundance of data, the detection of redundancies is of utmost importance as these can make later stages in big data analysis more lightweight. Clustering is an effective means to accomplish this. Here one would make the cluster boundaries sufficiently tight such that the resulting cluster centers serve as viable representatives for their respective sub populations. This boundary would be determined by the domain experts.

While GPUs can clearly accelerate preprocessing, the response will not be immediate but can possibly take hours or more. The best possible compromise is to give the user a glimpse of the final result – a partial result that can convey a good hint on what to expect when all is done. In computer graphics there is the concept of progressive rendering or refinement [2]. It is an approximation of the final rendered image in cases when the full resolution image will take a long time to compute. We have worked towards a solution into this direction for big data clustering, Necessitated by our domain application – the analysis of data acquired from a single particle mass spectral analyzer – we have focused on correlation clustering, but we could handle spatial distances, such as Euclidean, just as well.

To communicate these evolving results, visualization is an effective means. Here we have opted for a display that uses Multi-Dimensional Scaling (MDS) [11] to generate a dynamic 2D overview display of the emerging clustering results. Visual hints are given that allow 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

Two recent papers by Wong and colleagues list the top ten challenges in extreme-scale visual analytics [14][15]. Among these were user-driven data reduction, scalability and multilevel hierarchy, and data summarization. We achieve data reduction via our tight clustering framework which the user is able to drive and appreciate by visualizing the dynamic MDS display as a data summary. Then, once the clustering is finished, we can construct a hierarchical decomposition by merging the clusters into a tree (a dendrogram) using a correlation metric [9][12][18].

Our algorithm is a clustering scheme but its main purpose is data decimation. Clustering of big data is often done with the k-means algorithm. It 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 typically continue until the total sum of errors falls below some minimum. The k-means algorithm is not guaranteed to converge to the global minimum – typically multiple runs are taken, also with different numbers of *k*.

To parallelize *k*-means on distributed architectures interconnected via MPI/OpenMP, a typical approach is to partition the *N* data points onto the *P* processors. In this case each processor runs step 1 and 2 on its local data, and the global *k* means are found by averaging the local ones. This step can occur in parallel as well by using a map-reduce approach [3][22]. Here all mappers distribute their local *k* means to a set of *P* reducers which perform the averaging in parallel. Following, the reducers send the global *k* means back to the mappers for a new iteration. As such, this approach requires two communication steps per iteration but offers parallelism at every stage of the distributed algorithm.

Alternative to map-reduce, in another less parallel approach each processor broadcasts its local set of *k* means to all other processors which then all compute the global set locally. This approach requires only one communication step but offers less parallelism. It has found use, for example, in the P2P algorithm by Datta et al. [4] where a set of remote workstations are connected via TCP/IP.

Finally, k-means has also been accelerated on GPUs [1][6][8][17] using CUDA. Most approaches typically only parallelize step 1 but not step 2 since the number of clusters has been found to be too low for parallelization. We also use GPUs and CUDA but our purpose is not standard k-means where clusters can have any extent as long as they do not overlap with other clusters. Rather, in our algorithm clusters cannot have an extent greater than a preset threshold, in a correlation context. This makes a direct comparison to the existing, more general work difficult.

## III. Background

The clustering framework presented here is part of a larger visual analytics system that we have developed with a group of aerosol scientists over the past ten years [9][12][18][19][21]. The data acquired by a state-of-the-art single particle mass spectrometer, termed SPLAT II, are composed

(a) (b)

Fig. 1: Deployment of SPLAT II in a flight campaign.

of 450-dimensional mass spectra of individual aerosol particles (see [20] for more detail). SPLAT II can acquire up to 100 particles per second at sizes between 50-3,000 nm with a precision of 1 nm. It can be used for atmospheric chemistry to understand the processes that control the atmospheric aerosol life cycle. This is important, for example, in finding the origins of climate change, by uncovering and helping to model the relationship between atmospheric aerosols and climate. Other applications are fundamental science, nanotechnology, characterization of engine emission, and national security. Fig. 1a shows SPLAT II operated by the collaborating scientist in-flight in the Arctic aboard a Convair-580 research aircraft. Fig. 1b shows various sensor probes mounted on the aircraft wing.

The overall goal is to build a hierarchy of particles based on their spectral composition that can be used in subsequent automated classification of new particle acquisitions, either back in the lab or directly in the field. The tools we developed to create this hierarchy tightly integrate the scientist into this process. Our system provides a variety of interaction capabilities that allow the scientists to delineate particle clusters directly in high-dimensional (450-D) space – a process which we refer to as *cluster sculpting*. Interactive tools for this process are strongly needed since the data are extremely noisy and fully automated clustering tools do not return satisfactory results.

Fig. 2 shows the interface (called *SpectraMiner*) with a complete particle hierarchy in form of a radial dendrogram [9]. The outer ring has the leaf nodes which make up the set of particles. These are then merged into higher level nodes based on their mutual correlation. A heap sort algorithm

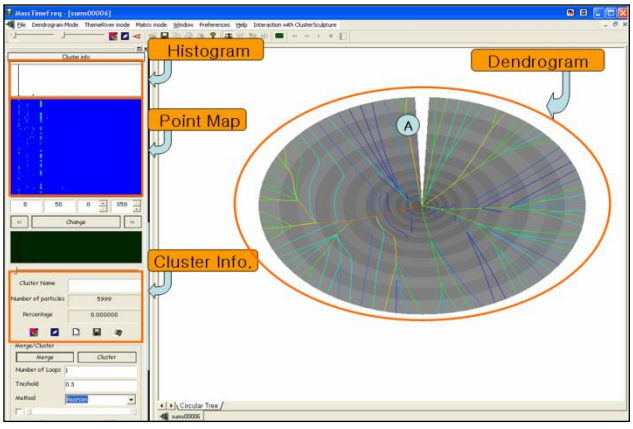



Fig. 2: The SpectraMiner interface.

merges the currently most correlated pair of nodes until reaching the root at which all nodes have been merged. Correlation serves as a distance metric here since we are primarily interested in particle composition ratios (mixing state) and not in absolute numbers. Since SPLAT II can acquire 100 particles/s, the number of particles gathered in a single acquisition run can easily reach 100,000 – it would take just a 15 minute time window.

Even 100,000 is a large number of points to compute the classification tree from and so we have always relied on correlation 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 in the large campaigns, such as the more recent one in the Arctic (Fig. 1), the runs 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 [16] 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 correlation clustering at this level of magnitude, and this necessitated a high-performance GPU solution.

## IV. Our Approach

The algorithm we present is not necessarily a new clustering algorithm; but a modification of an existing incremental clustering algorithm, used in [9], to make it more amenable to GPU acceleration. The incremental *k*-means algorithm has the desirable property of selecting a good value for *k* (i.e. the value of *k* does not need to be predetermined). The way the algorithm is structured, however, does not map very well to a parallel implementation.

### A. INCREMENTAL K-MEANS

Figure 3 shows pseudo code for the incremental algorithm of [9]. It starts by selecting the first point from a dataset of *N* points and making it the initial cluster center. The next point is then selected and compared to the initial cluster center using some distance metric. If this point is close enough to the cluster center (i.e. within some threshold), it is placed into the cluster. The center is then updated by taking the average of the coordinates of each point in the cluster. If the point is too far from the cluster center (i.e. outside of some threshold), then the point becomes a new cluster center. The next point in the dataset of *N* points is then selected and compared to the two cluster centers. This process is repeated until all points have been clustered.

As this algorithm is running, it uses some heuristics to identify clusters that will likely be outliers. More specifically, it will keep track of small clusters which have not been updated for a while, and mark them as outliers. A second pass is then performed to re-cluster the points in the "outlier" clusters. During the first pass, each point is only compared to the cluster centers that came before it. In the second pass, the outlier points can be clustered into any of the cluster centers that were formed during the first pass.

```
Make the first point a cluster center
While number of unclustered points > 0
    Pt = next unclustered point
    Compare Pt to all cluster centers
    Find the cluster with the shortest distance
    If(distance < threshold)
        Cluster Pt into cluster center
    Else
        Make Pt a new cluster center
    End If
End
```

Fig. 3: Incremental clustering algorithm pseudo code.

### B. PARALLEL INCREMENTAL K-MEANS

When applied to a big data setting, the incremental algorithm can become quite slow. The points toward the end of the dataset will have to be compared against all of the cluster centers that came before it. This becomes extraordinarily compute intensive; especially when the points are of high dimensionality and the cost of computing the distance metric takes a non-negligible amount of time.

The incremental nature of this algorithm makes it unclear how to efficiently map it to GPUs. One approach we considered was to parallelize over the cluster centers. In this approach, a point would be compared to each cluster center in parallel. Then a parallel reduction operation would take place to find the closest cluster center. There are a few problems with this approach. Toward the beginning of the algorithm, when there are not many clusters, the GPU would be highly underutilized. Also, we would iterate through all the points in a sequential fashion. We have found, in our experience, that it is better to parallelize over the largest parts of an algorithm, and since there is an order of magnitude more points than clusters, this algorithm contradicts this rule of thumb. Finally, this strategy still requires that we perform a second pass to re-cluster outliers.

Figure 4 presents the parallelization strategy we adopted, in the form of pseudo code. It follows the original sequential algorithm described in the previous section until we have *C* cluster centers chosen from the pool of unclustered points – *C* was determined through empirical evaluation to yield the best run time performance. Then we compare, in parallel, all points to each of the *C* cluster centers. Each point is either labeled with the cluster center that it belongs to, or it is labeled as unclustered. Essentially, we are performing the parallel *k*-means clustering algorithm described in [1] where *k*=*C* and the criteria for inclusion into a cluster is that the distance between a point and a cluster is below a user defined threshold. The *C* cluster centers are then updated in parallel. Following, all points are compared, again in parallel, to the updated cluster centers. This process repeats until the points converge, or a predefined number of iterations, Max_iterations, have been performed.

```
Make the first point a cluster center
While number of unclustered points > 0
    Perform sequential k-means until C clusters emerge
    Num_Iterations = 0
    While Num_Iterations < Max_iterations
        In Parallel: Compare all points to C centers
        In Parallel: Update the C cluster centers
        Num_Iterations++
    End
    Output the C clusters
End
```

Fig. 4: Our parallel incremental clustering algorithm

We then record the *C* cluster centers to an output buffer, and repeat the sequential algorithm to obtain a new set of *C* cluster centers, followed by another parallel clustering step using these new *C* centers.

This strategy essentially merges the incremental *k*-means algorithm with a parallel implementation of the traditional *k*-means algorithm, where *k*=*C*. The advantage here is that the sequential part of the algorithm never compares a point to more than *C* cluster centers. Another advantage to this approach is that we no longer need a second pass. Since every point is being compared to every cluster, a second pass would be unnecessary.

We chose C=96 through experimentation. We found that 96 clusters gave a good balance between CPU and GPU utilization. Conversely, with more than 96 clusters, the algorithm would become CPU bound. With less than 96 clusters, the GPU would be underutilized. It was also important that the number of clusters be a multiple of 32 to avoid divergent warps. See Section 5 for more detail. Finally, the limit *Max_iterations* was also determined experimentally. We found a value of 5 to be effective, but note that most iterations converge much earlier.

### C. CLUSTER VISUALIZATION

For the visualization of the evolving high-dimensional data clusters we aimed for a display that can show these clusters intuitively. A low-dimensional embedding into the 2D plane, as provided by MDS, is such a visualization and we chose Glimmer MDS [10] for this purpose. Glimmer MDS is a multilevel dimension reduction and visualization technique. It 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. It consists of three main stages: (1) restriction, (2) interpolation and (3) relaxation. The restriction stage is to build the multilevel hierarchy by randomly assigning each point to a certain level. The interpolation phase is to compute the current level's points' coordinates by interpolating them into the lower dimensional space where the embedding of all points from the previous level is fixed. Finally, the relaxation stage is to take all points into consideration and move them around to achieve a globally minimum stress value. For the interpolation and relaxation, Glimmer MDS adopts a stochastic force algorithm. The main advantage of this algorithm is that the final embedding of a point is only decided by a small number of points. Even though the set of those referential points changes at every iteration, its size remains unchanged and thus the computation cost is also fixed. This property makes Glimmer MDS quite amenable to GPU acceleration.

Next we need to finalize what to visualize. Since the number of points is massive, if we visualize all of them the visualization will become cluttered. Since after clustering the points each cluster center is a fairly tight representative of its members, it is reasonable to only visualize the cluster centers. Further, we also only visualize significant clusters – those that have more than *M* members – we chose *M*=10. A histogram visualizes the size distribution of the clusters and a transfer function maps the number of members of each cluster to the color of its displayed cluster center.

The range of the number of members in each cluster can be vary widely (in our application, from 1 to 317,786). Moreover, the cluster sizes may not be evenly distributed across this range. Therefore a linear transfer function would not effectively capture the difference. By applying a piecewise transfer function, however, we can exercise more control over how points are colored. Since we only have one parameter (i.e. the number of members in each cluster) that distinguishes the cluster centers, we link it only to its color saturation change. We use a color map from white to blue; with small clusters mapped to mostly white and large clusters to saturated blue. By incorporating this visualization component into our parallel incremental clustering algorithm we can provide a streaming experience where we initially visualize the first *C* cluster centers, and update the view with new cluster centers as they are formed.

## V. LOW LEVEL DETAILS

We implemented the parallel clustering algorithm described in the previous section using CUDA. The distance metric we used in this implementation is the Pearson distance metric $d_{xy}=1-\rho_{xy}$ where $\rho_{xy}$ is the Pearson correlation coefficient. We will now describe how we compare each point to *C*=96 cluster centers in parallel.

For our CUDA function, we launch *N/32* thread blocks of size 32 × 32; where *N* is the number of points. Each thread block compares 32 points to 96 cluster centers. The *x* coordinate of each thread tells it what point it will be operating on; while the *y* coordinate tells it what cluster center the point will be compared against – each thread processes three cluster centers. The points and the cluster centers are stored in memory as two matrices (see figure 5). Both matrices have an *x* dimension of 450 –the number of bins in the mass spectrum. The naïve approach would be to map each thread to a point and have it iterate along the *x* dimension of the point and its corresponding cluster center

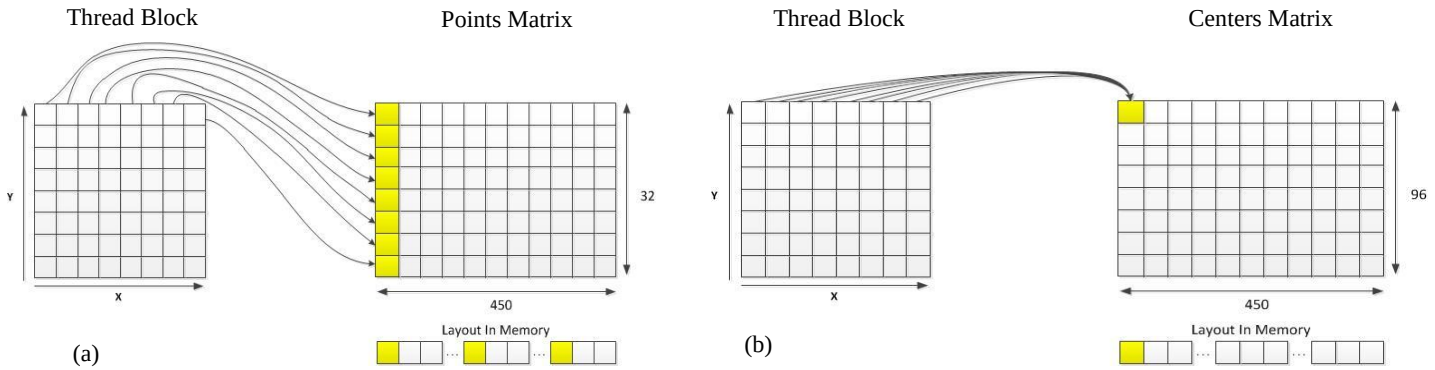



Fig. 5: Access pattern for naïve GPU clustering. (a) Per warp access pattern of the "Points" matrix. (b) Per warp access pattern of the "Centers" matrix.

to compute the Pearson distance. This, however, leads to an un-coalesced access pattern. We can see in figure 5(a) that the access pattern is quite sparse for the memory layout of the "points" matrix. This results in 32 separate data transfers for each warp. The access pattern for the cluster "centers" matrix is better, as seen in figure 5(b). Since every thread in a warp is accessing the same data at the same time, this results in one data transfer per warp.

The latency due to the poor memory access is the primary bottleneck of this application. To improve the throughput of the access to the points matrix, we load a 32 × 32 block of the points matrix into shared memory in a coalesced fashion (i.e. transpose the access pattern in figure 5(a)). Once the data is in shared memory, we can use almost any access pattern and still have a high throughput. This technique could also be used to improve the throughput of the cluster centers matrix. Unfortunately, there is not enough shared memory to handle both matrices without a drop in occupancy, and hence, overall performance.

Figure 6 shows the pseudo code for our GPU accelerated clustering algorithm. The optimizations presented in this section are implemented in the *PearsonDist* function and are omitted for brevity. Each thread computes the Pearson distance between a point and three cluster centers – resulting in 96 cluster centers per thread block. Increasing the per-thread workload in this way helps the CUDA scheduler hide latency from costly memory transfers. Although the choice of 96 clusters was influenced by the dataset, what is important is that performance is often improved by increasing the per-thread workload. The *PearsonDist* function returns the cluster that is closest to the thread's point, and the corresponding distance value. Since every column in the thread block operates on a separate point, and every thread within a column operates on the same point, an intra-column parallel reduction takes place to find the cluster with the minimum distance. By doing this, we arrive at one cluster center and one distance value per column. We can then compute the new cluster centers by using a parallel segmented scan algorithm [13]. Computing the cluster centers, however, is small enough that a simple implementation using the "atomic add" function is suitable.

```
c1 = Centers[tid.y]  // First 32/96 loaded by thread block
c2 = Centers[tid.y + 32] // Second 32/96 loaded
c3 = Centers[tid.y + 64] // Final 32/96 loaded
pt = Points[tid.x]

[clust, dist] = PearsonDist(pt, c1,c2,c3)
[clust, dist] = IntraColumnReduction(clust,dist)

//first thread in each column writes result
If(tid.y == 0)
    Points.clust[tid.x] = clust
    Points.dist[tid.x] = dist
End If
```

Fig. 6: CUDA kernel for clustering. Each thread block compares 32 points to 96 clusters (i.e 3 clusters per thread)

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

## VI. Results

We implemented our parallel *k*-means clustering algorithm on a server with 4 Tesla K20 GPUs. We also implemented the sequential algorithm presented earlier. The user-defined threshold was set to a Pearson distance of 0.3 throughout.

We measure clustering quality using the Davies-Bouldin (DB) index [5] (equation 1). It is a measure of intra-cluster distance, σ, as well as the distance between cluster centers, $M_{ij}$ for the total number of clusters, $n$. 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*, 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 sub-threshold levels affect quality. Unexpectedly, clustering with no sub-threshold 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).

Intuitively, this makes sense. Because the points are of high dimensionality, it is likely that if *k* points find clusters that are closer, they will be in *k* separate clusters. Within each cluster, a single point will have a small impact on the intra-cluster distance. The cluster that is losing *k* 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 *k* updated clusters will have a smaller inter-cluster distance.

Figure 7 shows how the timings scale with increasing datasets for the different implementations. The graph compares the sequential to the parallel implementation with differing values for the sub-threshold. Here we can see the effectiveness of our sub-thresholding optimization. As the dataset increases, the sub-thresholding optimization helps to suppress the explosion in compute time required.

We also implemented a multi-GPU solution. This solution is very similar to the parallel algorithm presented earlier. The only difference is that the dataset is distributed to each GPU equally. One step is added after the GPU clustering call to merge per-GPU cluster centers. The rest of the algorithm remains unchanged. We can see, from figure 7, that the compute time required grows at a super-linear rate with respect to the size of the dataset (i.e. doubling the size of the dataset requires more than double the compute time). Another interpretation of this graph is that halving the size of the dataset requires less than halve the compute time.

This super linear growth rate is what makes this algorithm a good candidate for multi-GPU acceleration. We can see in figure 8 that using two GPUs requires less than half the compute time when compared to one GPU. This level of speed-up is rarely seen when implementing a multi-GPU solution. We observe it here because each GPU is working on half the dataset. We do not, however, observe a similar increase in performance when moving from two to four GPUs. This suggests that there is a diminishing rate of return in the multi-GPU approach.

TABLE I: DB SCORES OF CLUSTERS

| Size | Seq. | Par. | 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.

In figure 9 we show some visualizations generated during the incremental *k*-means clustering. For each row, the top scatterplots are the MDS layouts of the cluster centers and the bottom histograms show the distributions of the number of members of all clusters. The two numbers under each figure indicate the number of significant cluster centers being visualized (i.e. having more than 10 members) and the total number of clusters processed.

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 *k*-means algorithm. Some clusters are actually close but not close enough to form one single cluster. Second, the ratio of the significant clusters (i.e. number of members more than 10) 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, 4,000 of these are significant (>10 points), while for the second 165,984 clusters only 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.

Lastly, we note that the clusters appear rotated from plot to plot. This is purely due to the MDS layout algorithm that

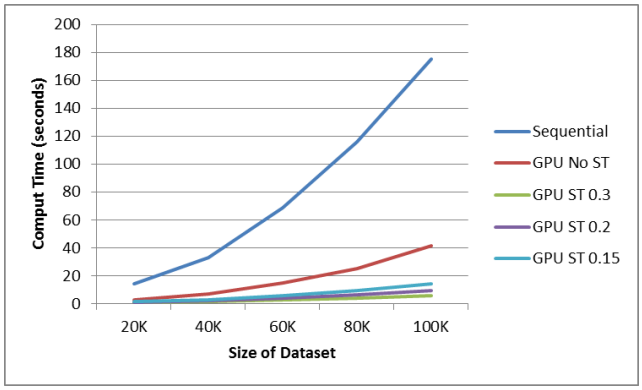



Fig. 7: Timings of parallel and sequential incremental k-means algorithm

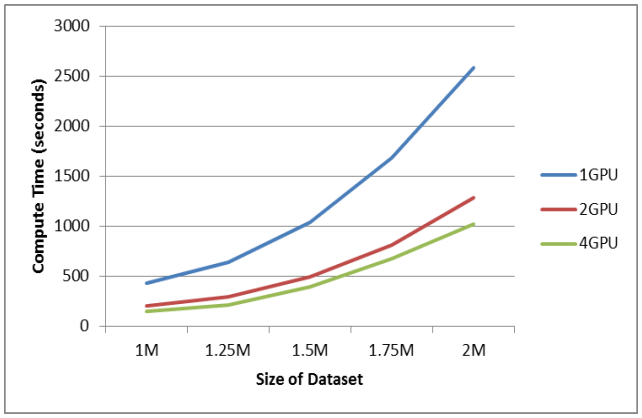



Fig. 8: Timings of the multi-GPU parallel incremental k-means algorithm (ST: 0.15)

does not prefer certain layout orientations. Future work will use anchoring to maintain the same orientations throughout.

## VII. Conclusion

We have presented a novel approach to parallelizing a sequential incremental *k*-means clustering algorithm. We observed an order of magnitude speed-up over the sequential version. Our results also show that the compute time required for parallel implementation grows at a slower rate compared to the sequential implementation. The compute time required to cluster the entire 8 million points dataset on four GPUs was just under two hours; whereas the sequential implementation required several days for this.

Future work will include an out of core approach to handle larger dataset sizes. At 8 million points, our system is at the limit of what it can store in RAM. We would also like to examine the effects of load balancing techniques on performance. In the multi-GPU approach, points are not necessarily removed from the datasets on each GPU in a balanced way. One load balancing solution would be to redistribute points onto each GPU after every call to the parallel clustering function. Additional experimentation is needed to determine if the latency of this extra memory transfer is small enough to reach a net gain in performance.

For completeness, a multi-core CPU solution should also be analyzed. The sequential incremental algorithm is non-parallelizable and adapting this algorithm to expose parallelization comes at a cost of increasing the total amount of work (i.e. comparing every point to every cluster versus every point to a subset of clusters in the sequential version). GPUs are powerful enough to overcome this extra work and achieve speed-ups over the sequential implementation. It is unclear whether a multicore solution will be able to overcome this extra work as well. A more complete analysis is left for future work.

We believe providing visual feedback while the clustering algorithm is running can be very helpful. It can allow users to fine tune the clustering parameters and process. In the context of big data (where processing takes a long time) streaming visualizations can provide users with immediate feedback. Visualizing results is also more intuitive than raw text data and can increase the chance of spotting patterns that might otherwise go unnoticed.

## Acknowledgments

This work was partly supported by NSF grant IIS-1117132, by the Ministry of Korea Knowledge Economy, and DOE STTR grant DE-SC0009678. Partial support was also provided by the US Department of Energy (DOE) Office of Basic Energy Sciences, Division of Chemical Sciences, Geosciences, and Biosciences. Some of this research was performed in the Environmental Molecular Sciences Laboratory, a national scientific user facility sponsored by the DOE's OBER at Pacific Northwest National Laboratory (PNNL). PNNL is operated by the US DOE by Battelle Memorial Institute under contract No. DE-AC06-76RL0.

## References

[1] H. Bai, et al. "K-means on commodity GPUs with CUDA." *IEEE World Congr. on Comp. Sci. & Info. Eng.*, pp. 651–655, 2009.

[2] M. Cohen, S. Chen, J. Wallace, D. Greenberg," A Progressive Refinement Approach to Fast Radiosity Image Generation." *Computer Graphics (Proc. SIGGRAPH*, 22(4), pp. 75–84, 1988..

[3] C. Chu,, S. Kim,, Y. Lin. et al., "Map-Reduce for Machine Learning on Multicore," *NIPS*, 19:281-288, 2007.

[4] S. Datta, C. Giannella, H. Kargupta, "Approximate Distributed K-means Clustering Over a Peer-to-Peer Network." *IEEE Trans on. Knowledge and Data Engineering*,, 21(10):1372-1388, 2009.

[5] D. L. Davies, D. W. Bouldin, "A Cluster Separation Measure," *IEEE Trans.on Pattern Analysis and Machine Intelligence*, 224-227, 1979

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

[7] B. Howe. *Introduction to Data Science*, Coursera Course (06/15/13)

[8] M. Hsu, R. Wu, B. Zhang, "Uniformly Fine-grained Data Parallel Computing for Machine Learning Algorithms," in *Scaling Up Machine Learning*, R. Bekkerman, M. Bilenko, J. Langford, eds, Cambridge University Press, 2012.

[9] P. Imrich, K. Mueller, D. Imre, A. Zelenyuk, W. Zhu, "Interactive Poster: Visual Datamining with the Interactive Dendogram," *IEEE Information Visualization Symposium*, Poster Session, October 2002.

[10] S. Ingram, T. Munzner, M. Olano, "Glimmer: Multilevel MDS on the GPU," *IEEE Trans Vis. & Comp. Graph., 15(2):249-261, 2009.*

[11] J. Kruskal, "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis," *Psychometrika*, 29(1):1–27, 1964.

[12] E. Nam, Y. Han, K. Mueller, A. Zelenyuk, D. Imre, "ClusterSculptor: A Visual Analytics tool for high-dimensional data," *IEEE Symp. Visual Analytics Science & Technology (VAST)*, pp. 75-82, 2007.

[13] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, "Scan Primitives for GPU Computing," *Proc. Graphics Hardwaree*, 97-106, 2007

[14] P. Wong, H. Shen, C. Johnson, C. Chen, R. Ross, "The Top 10 Challenges in Extreme-Scale Visual Analytics," *IEEE Computer Graphics & Applications*, 32(4): 63 – 67. 2012.

[15] P. Wong, H. Shen, C. Chen, "Top Ten Interaction Challenges in Extreme-Scale Visual Analytics*," Expanding the Frontiers of Visual Analytics and Visualization*, Springer, pp. 197–207, 2012.

[16] R. Zaveri et al. "Overview of the 2010 Carbonaceous Aerosols and Radiative Effects Study (CARES)," *Atmos. Chem. Phys*. 12:7647-7687, 2012.

[17] M. Zechner, M. Granitzer. "Accelerating k-means on the graphics processor via CUDA." *Proc. IEEE INTENSIVE*, pp. 7-15, 2009.

[18] 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," *International Journal of Mass Spectrometry*, 258:58-73, 2006.

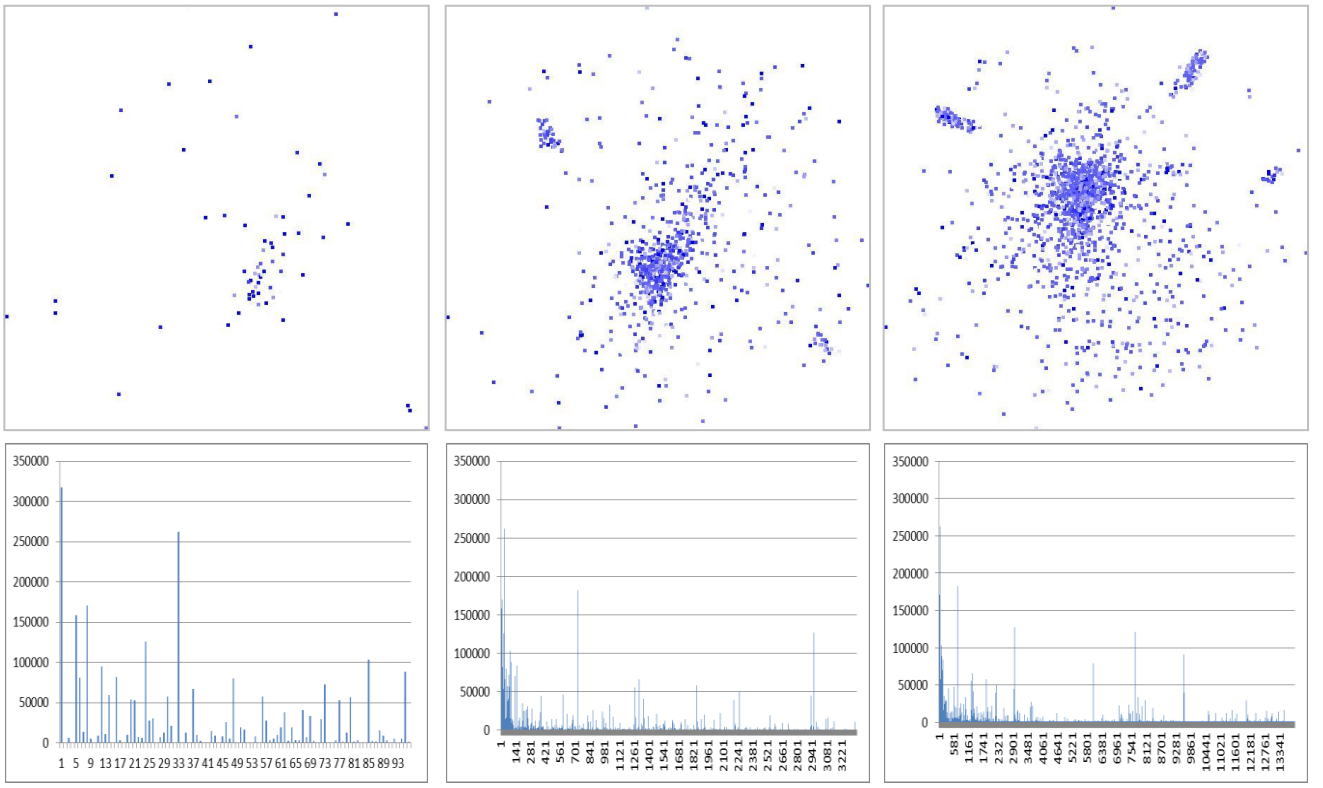
[19] A. Zelenyuk, D. Imre, E. Nam, Y. Han, K. Mueller, "ClusterSculptor: Software for Expert-Steered Classification of Single Particle Mass Spectra," *Intern. Journal of Mass Spectrometry*, 275(1-3):1-10, 2008.

[20] A. Zelenyuk, J. Yang, D. Imre, E. Choi, "SPLAT II: An aircraft compatible, ultra-sensitive, high precision instrument for in-situ characterization of the size & composition of fine & ultrafine particles," *Aerosol Sci. Technol*. 43: 411-424, 2009.

[21] Z. Zhang, X. Tong, K. McDonnell, A. Zelenyuk, D. Imre, K. Mueller. "An Interactive Visual Analytics Framework for Multi-Field Data in a Geo-Spatial Context" *Tsinghua Science and Technology on Visualization and Computer Graphics*, 18(2), April, 2013.

[22] W. Zhao, H. Ma, Q. He, "Parallel K-Means Clustering Based on MapReduce," *Cloud Computing, Lecture Notes in Computer Science* Volume 5931, pp. 674-679, 2009.

[23] http://www.kdd.org/blog/age-big-data-kdd-89-kdd-2012 (07/29/13)



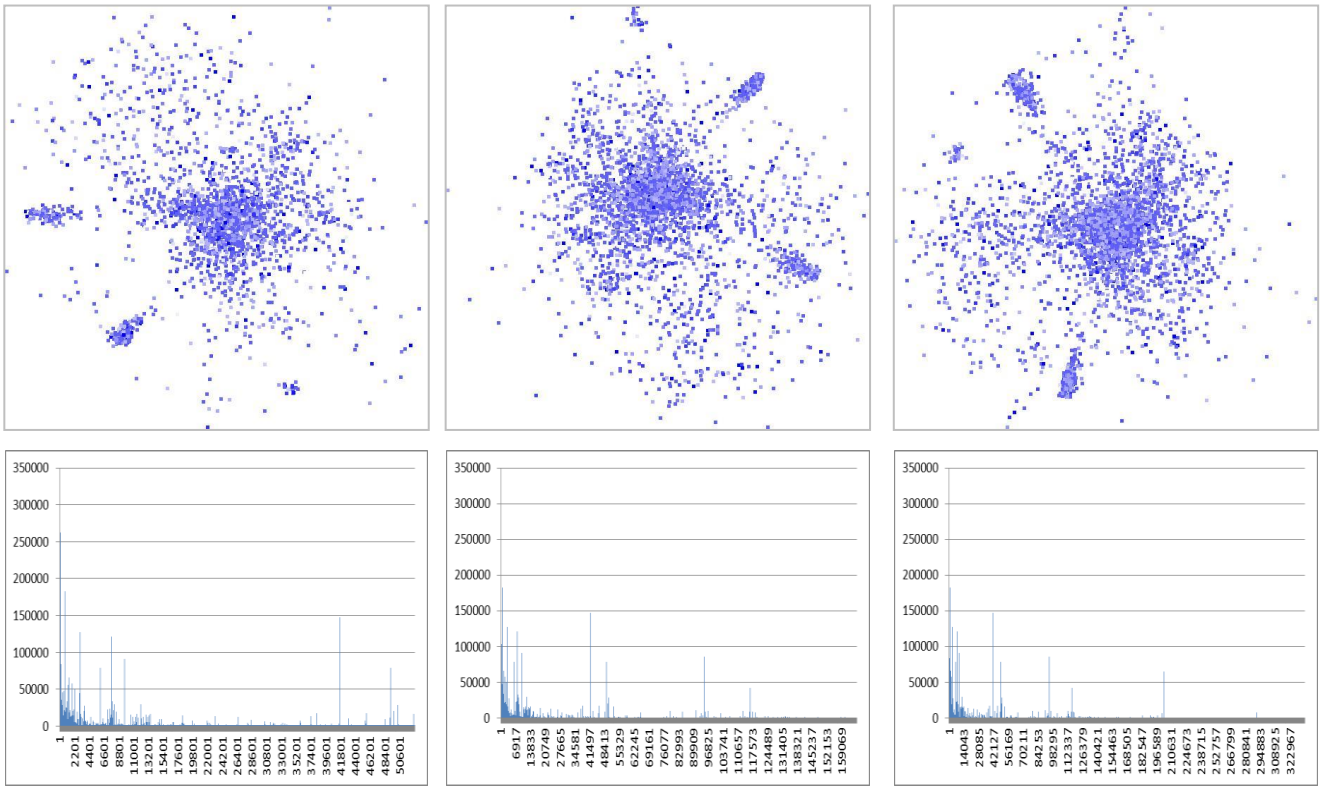




Figure 9: Cluster visualization. The number below each image is the number of cluster centers being visualized vs. the total number of cluster centers. The centers not visualized are those that have a population of less than 10 members.