

Towards Flow Cytometry Data Clustering on Graphics Processing Units

Jeremy Espenshade¹, Doug Roberts¹, James Cavenaugh², and Gregor von Laszweski¹
 Rochester Institute of Technology, Rochester, NY¹
 University of Rochester, Rochester, NY²

Abstract—Like many modern techniques for scientific analysis, flow cytometry produces massive amounts of data that must be analyzed and clustered intelligently to be useful. Current manual binning techniques are cumbersome and limited in both the quality and quantity of analysis produced. To address the quality of results, a new framework applying two different sets of clustering algorithms and inference methods are implemented. The two methods investigated are fuzzy c-means and minimum description length inference and k-medoids with BIC. These approaches lend themselves to large scale parallel processing. To address the computational demands, the Nvidia CUDA framework and Tesla architecture are utilized. The resulting performance demonstrated 1-2 orders of magnitude improvement over an equivalent sequential version. The quality of results is promising and motivates further research and development in this direction.

I. INTRODUCTION

Flow cytometry is a technique for elucidating the phenotypes of cells in a suspension. It is a mainstay technology used in immunology, although other fields also use it. The process involves allowing fluorescently dyed antibodies to bind to proteins (antigens) on the surface of the suspended cells. Then using a technique known as hydrodynamic focusing, the stained cells pass through a laser beam one at a time, which excites the fluorophores that are indirectly attached to the antigens of interest. (Modern instruments use sequential lasers with a delay time to attach information from subsequent laser excitations to information from the first laser. Presently 18-color instruments are available, which also have information from light scatter as well.) The resulting fluorescence and light scattering data are measured by a set of sensors and recorded as a vector of d -length. Such a vector is generated for each event, which is typically a cell passing through the laser beam. Cells pass through the beam at the rate of thousands each second and a data file for even a single stained sample may typically involve up to one million events or more. This large data set is stored in an FCS file format (for flow cytometry standard). Finding clusters in these large and high-dimensional data sets is an application ideally suited for massively parallel computation.

The Center for Biodefense Immune Modeling at the University of Rochester is engaged in immunology research that includes the use of flow cytometry for cellular analysis. The level and value of such analysis is currently limited by the manual techniques involved, and an opportunity exists to apply novel methodologies leveraging cyberinfrastructure and current parallel computing architectures. Specifically, Nvidia's CUDA

framework for scientific computation on parallel streaming processors presents a promising opportunity for low cost, high performance analysis [1].

II. BACKGROUND

Flow cytometry allows researchers to identify and characterize populations of cells of interest by their co-expression of antigens which serve as markers. Currently, this is done using manual filtering where the researcher draws bins (called gates in the flow cytometry literature) around clusters of data in successive two dimensional histograms. This approach is essentially unchanged from twenty years ago and has a number of disadvantages. Variability between experienced immunologists can be as high as 10-fold for difficult data sets (unpublished research from the University of Rochester's David H. Smith Center for Vaccine Biology and Immunology). As d increases, the number of histograms increases combinatorially, making the data analysis process more difficult and tedious. There are presently no widely used standards in flow cytometry data analysis, and gates are not reported. It is therefore impossible to accurately reproduce other's work from only the raw data, and a sensitivity analysis using slight variations in the bin positions is impractical. The outcome of this time-consuming manual process is a result that is both imprecise and not very accepting of modification. Furthermore, manual sequential bivariate binning does not make full use of the multivariate nature of the data and is not conducive to making theoretically sound statistical inferences from these data sets.

Clearly an automated process for identifying cells of interest would be advantageous. Whether a sequential bivariate approach or a fully multidimensional approach is used, the problem is essentially one of finding a suitable clustering of the data. Clustering can either be hard or soft (fuzzy); hard clustering requires every datum to belong to exactly one cluster. The current practice of sequential bivariate gating is a manual version of hard clustering. Fuzzy clustering allows each datum to belong to different clusters with different probabilities of membership (or viewed another way, with different mixture amounts from underlying archetypal distributions). By representing an event's cluster membership as a set of probabilities, one for each possible cluster, single events can be included in multiple clusters and also exert influence on the cluster location based on how closely associated they are with the cluster in question. The benefit of this is most obviously realized by the marginalization of outliers. Since cluster centers are essentially a mean of the member events, outliers have

a tendency to shift the calculated center away from the logical center without fuzzy clustering. Fuzzy clustering is a much better characterization of the underlying biology than is hard clustering. By varying a cutoff for probability of membership in a cluster, it is trivial to convert fuzzy clustering to hard clustering, which makes it easy to do a sensitivity analysis. It is also possible to go in the reverse direction: one can soften a hard clustering by a function which maps the distance of each datum from each cluster's center to a probability of belonging to that cluster.

A remaining difficulty is determining the number of clusters to use. Center-based clustering algorithms, like the fuzzy c-means or k-medoids, require some initial number of clusters. If too many clusters are chosen, the results may be duplicated or muddled by separating logically singular clusters, and if too few clusters are chosen, meaningful data will be lost due to combination or elimination of distinct clusters. To solve this problem, Selb et al [2] integrated c-means into a Minimum Description Length (MDL) framework. The MDL Principle states that the more similarity that exists in a data set, the more the data can be compressed. Learning is then equated with compression. An MDL-framework then works to identify the ideal set of reference vectors, or cluster centers in this context, that describe most of the data while minimizing the number of such vectors. MDL is less well suited to a hard clustering method like k-medoids, but other inference methods like Bayesian Inference Criterion are potential candidates.

As with any fuzzy clustering algorithm, and extended with the use of MDL, several meaningful parameters governing relations between variables can have a large impact on the final result. Such parameters include the degree of fuzziness, or how much influence cluster members exert on the cluster center, fuzzy membership threshold, and the MDL weighting parameters that govern the relative weight data point description, reference vector description, and error. Such parameters are impossible to optimally set a priori and vary between data sets. Therefore, multiple analysis with differing parameters are potentially useful as well.

A. FCS Data Analysis

Figure 1 illustrates the statistically formulated cluster analysis work flow for FCM data. It is a multi-step process with some optional steps. First the data is read from an FCS file. Header information and metadata are ignored, leaving only raw flow data. The extracted data may be filtered in order to restrict attention to the regions of parameter space known a priori to be interesting and thereby to shorten the data upon which clustering will actually be performed. Ideally this would involve an automated approach as well, probably based on image analysis of the forward versus side scatter plot.

Second, a decision as to follow a sequential bivariate approach or a simultaneous multivariate approach needs to be made. The former is current practice, but could be automated using image processing algorithms such as found in Matlab. It offers the advantages of familiarity and ease of use for finding populations of cells which are known in advance to be of interest. The latter approach is more conducive to exploring

the data for unanticipated findings and is more suited for formal statistical inference. It is also the approach followed in this paper. Third, the data may need to be transformed, and compensation may need to be applied to reduce the effect of fluorescence spillover from a fluorophore maximally excited in one channel to other channels. Fourth, the data may need to be transformed, as for example by log, biexponential, or logicle transformation [3]. Compensation and transformation are especially important for image based approaches such as sequential bivariate gating. Fifth, a distance measure needs to be decided upon, Euclidean being the most common. Sixth, the (possibly transformed) data may optionally be standardized and normalized. Seventh, the data are then clustered using any of the many possible clustering algorithms. After clustering the data a statistical summary should be prepared and the results should be visualized graphically [4]. Finally, the process is repeated for other samples in the experiment and then statistical and biological inferences can be made.

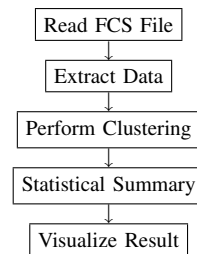


Fig. 1. Objective and Automated Cluster Analysis Workflow [4]

III. PARALLEL COMPUTING FOR FLOW CYTOMETRY

Given the large amount of data, the complexity of the algorithms involved, and the need for many computations on the same data set, a robust set of computing resources are needed. Fortunately, the problem is inherently parallel and its computation easily distributed across a number of resources. Besides the algorithmic design, the complexity of data allocation, optimal task sizing, and communication all remain difficult problems. As such, the biostatistician is unlikely to have the expertise and uncommitted time necessary to manage the computations at a computational resource level. There is therefore a need for an abstraction of these resources such that the researcher can focus on the conceptual challenges and receive results that are immediately useful.

For this reason the problem at hand is a good candidate for parallel computing infrastructures. The large amount of data that is produced while scanning individual cells, as they pass through the laser beam, should be able to be divided up and distributed across multiple processors or node on a grid. Once the data has been divided and distributed each processor can proceed to perform computations on its own chunk of data. This will reduce the amount of time needed to process the entire data set as compared against the amount of time that is needed to process the data on a single processor machine. Once each of the processors has finished performing computations their results will be combined and a final result will be generated. In the case of flow cytometry problem, each

processor will cluster its own set of data. Once completed the separate clusters need to be combined together in such a way that the entire dataset is clustered properly. Such a result can then be communicated back to the researcher while maintaining the abstraction that hides the aforementioned details.

For this project a set of clustering algorithms will be written that will be able to run on a CUDA enabled device. CUDA is discussed in further detail in the following section. The data being clustered for this project is basically a large grid. In CUDA this data grid can be divided up into several blocks. Each of the blocks contains a number of threads. Each of the threads will process their respective data in the grid in order to perform the clustering. This is essentially what would happen on a traditional grid system. However in CUDA the grid is a large number of threads that run on a single device, either a graphics card or a Tesla device.

IV. CUDA

In recent years, traditionally fixed-function graphics processors have transitioned into massively parallel stream processors capable of general purpose computation. Each Nvidia Tesla C870 has 128 processing units organized into 16 multiprocessors capable of handling thousands of threads, or separate streams of execution, concurrently. To manage thread creation, synchronization, and data allocation, Nvidia has developed and provided a set of APIs, compilers, and supporting libraries collectively referred to as the Compute Unified Device Architecture (CUDA). Applying data-parallel applications to the CUDA framework has been shown to provide performance on the order of hundreds of times faster than a single general purpose processor [5] [6] [7]. Furthermore, as the cost of a CUDA-enabled device is less than an individual workstation, the cost-performance ratio compared to a traditional cluster can be staggering. Given this potential, combined with the large computation requirements of the flow cytometry application detailed above, CUDA promises to be a valuable platform for investigation. To enable the reader to better understand the algorithmic and implementation details in subsequent sections, an overview of the architecture and programming model are provided in this section. For more details on the Tesla architecture [8] please consult the CUDA Programming Guide from Nvidia [9].

A. Memory Model

The memory is divided spatially between on-chip memory included in the GPU silicon and the graphics memory held in dedicated memory modules elsewhere on the circuit board. As one would expect, the access times vary greatly between these two memory locations. The difference can be as high as 1 cycle for on-chip memory compared to 400 - 600 cycles for off-chip memory [10].

The on-chip memory is divided by multiprocessor and consists of 16 KB of shared memory space and read-only caches for constant and texture memories. In addition to the shared memory and caches, each multiprocessor contains 8192 32-bit registers. The off-chip memory is organized into global,

local, texture, and constant memories, each of which have different functions. Constant and texture memories are read-only sections of memory that are cached as mentioned, thereby allowing repeated and spatially local memory accesses to read from the on-chip cache rather than the slow off-chip memory. Local memory may be allocated by individual threads and global memory is allocated by a host system. The entirety of off-chip memory is 1.5 GB for the Tesla card used.

B. Thread Model

As previously alluded to, the computation model enabled by CUDA is for the GPU to act as a co-processor for the main CPU, usually referred to as the host. To manage the many threads required to fully exploit this highly parallel architecture, a hierarchy of thread execution organization is required. The highest level of hierarchy is known as a *grid*, and represents all of the resources involved in the computation of a specified task, or kernel. The type of computation that constitutes a kernel is not rigidly defined, but it must be explicitly parallel in order to take advantage of the rest of the thread hierarchy and gain benefit from the hardware architecture. In typical usage, the CPU will specify a kernel to be computed and a grid to perform the computation.

A grid is then logically divided into a three-dimensional structure of thread blocks, each of which contains some number of threads organized into their own three-dimensional structure. While this appears complicated, the standard variables specifying the *blockId* and *threadId* allow logical thread organization. Each thread block controls a portion of the shared memory of a single multiprocessor and shares that space among all of the threads in that thread block, enabling inter-thread communication without expensive global memory accesses. Additionally, threads within a thread block may be synchronized through global synchronization points.

Between thread blocks, synchronization and data sharing are much slower and may require a return to host control. Because of this, thread blocks are typically quite independent of one another. Within a single thread block however, some additional restrictions are important to understand. These threads are organized into 32 thread groups called *warps* that physically execute concurrently. Each *warp* must fully implement a SIMD (single instruction, multiple data) program, meaning that there can be no divergent branches or other differing instructions based on the *threadID* within a warp or the warp will be split and parallelism reduced. Also of importance: the shared memory is divided into 16 banks that feed data to the executing threads, and bank conflicts (multiple threads attempting to read different values from the same bank) will result in slowed execution up to a factor of 16 in the worst case. Careful execution organization is therefore required to effectively manage the threads within a thread block.

The next section details the clustering algorithms employed.

V. ALGORITHMS

This section discusses some of the clustering algorithms applied to the flow cytometry problem. An explanation of each algorithm will be given as well as pseudo code for each

algorithm. For each algorithm an explanation will be given as to how the algorithm can be written such that it can run as a distributed program.

A. K-Medoids

K-medoids is a clustering algorithm that is related to the k-means algorithm. The k-medoids is a partitioning algorithm that divides the data set up into separate clusters. The algorithm also attempts to minimize the squared error which is the distance between points in the cluster and a point that is designated as the center (medoid) of a cluster. The k-medoids algorithm is a lot more resistant to outliers than the k-means algorithm. A medoid is considered an object of a cluster whose average dissimilarity to all the objects in a cluster is minimal [11].

The k-medoids algorithm functions by placing data into k clusters. k is a predetermined number that is chosen before the algorithm is executed. The algorithm functions as follows.

- 1) Randomly select k objects that will serve as the medoids
- 2) Associate each point in the data set with its most similar medoids using a distance measure (Euclidean distance, Manhattan distance, Minikowski distance, etc) and calculate the cost
- 3) Randomly select a nonmedoid object O
- 4) Replace a current medoid with the chosen non-medoid and calculate the cost again
- 5) If the new cost is greater than the old cost then stop the algorithm
- 6) Repeat 2 through 5 until there is no change in the medoid

The cost for each data point is calculated using Equation 1, Where x_i is the i^{th} data point in the data set and d is the size of the data set.

$$Cost = \sum_{i=1}^d dist(x_i) \quad (1)$$

In order to fuzzify the clusters, a membership value of each data point to every cluster (medoid) is calculated using equation 2.

$$P(x|m) = 1 - \frac{|x-m|}{\sum_{i=1}^k |x-m_j|} \quad (2)$$

Where x is a data point m is the medoid associated with the data point and m_j is the j^{th} medoid.

The Bayesian information criterion (BIC) [12] was integrated into the algorithm in an attempt to determine the best number of cluster for a given data set. The equation for the BIC is shown below.

$$BIC = n * \ln\left(\frac{RSS}{n}\right) + k * \ln(n) \quad (3)$$

Where n is the number of data points and k is the number of clusters being considered. RSS is the residual sum of squared errors which shown below.

$$RSS = \sum_{i=1}^n (x_i - m_j)^2 \quad (4)$$

Where n is the number of data points and x_i is the i^{th} data point and m_j is one of the medoids.

B. Fuzzy C-Means

K-means is a well known center-based clustering scheme [13] that performs hard clustering on the data by assigning each data point a membership the cluster who's center is closest to the data point. The cluster centers are then recalculated based upon the members of each cluster. Iteration stops once the change in cluster center is less than some epsilon value.

The benefit of k-means is in it's simplicity and rapid convergence to a reasonable solution. The limitations are that, as a hard clustering algorithm, it is strongly affected by scattered data outside of logical clusters. Among other techniques, filtering could be applied to the original data to reduce outliers and lessen their impact, but as additional computation steps are added to make up for limitations, the simplicity benefit is simultaneously diminished. Another important limitation is that the number of clusters must be specified a priori. As discussed in the background section, improper estimations can be very detrimental to the accuracy of the reported cluster centers, and FCS data in particular does not provide predictable numbers of clusters.

To address some of the limitations of K-means, fuzzy C-means was proposed by Dunn [14] and later refined by Bezdek [15]. Fuzzy clustering allows each data point to have a membership in every other cluster, with higher membership values being assigned to clusters closest to the data point. This approach has two primary advantages over k-means. It forces outliers to have less effect on the cluster centers by assigning a lower membership value to any particular cluster. It also mitigates the effect of starting with too many clusters for the data. While k-means may split a logical cluster into several distinct sections with cluster centers in each section, fuzzy c-means will converge on the center of logical clusters resulting in nearly duplicate results that are all close to correct. The algorithm is based on the minimization of the following function defining the error associated with a solution [16].

$$E_m = \sum_{i=1}^N \sum_{j=1}^C u_{ij}^p \|x_i - c_j\|^2, 1 \leq m < \infty \quad (5)$$

In Equation 5 p is any real number that is greater than one and defines the degree of fuzziness, u_{ij} is the membership level of event x_i in the cluster j , and c_j is the center of a cluster. The fuzzy clustering is done through an iterative optimization of Equation 5. Each iteration, the membership u_{ij} is updated using Equation 6 and the cluster centers c_j are updated using Equation 7.

$$u_{ij} = \frac{1}{\sum_{k=1}^C \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|} \right)^{\frac{2}{p-1}}} \quad (6)$$

$$c_j = \frac{\sum_{i=1}^N u_{ij}^p * x_i}{\sum_{i=1}^N u_{ij}^p} \quad (7)$$

Below is an outline of a fuzzy c-means algorithm.

- 1) Given the number of clusters, c , randomly choose c data points as cluster centers.
- 2) For each cluster, sum the distance to each data point weighted by it's membership in that cluster
- 3) Recompute each cluster center by dividing by the associated membership value of each event
- 4) Stop if there is minimal change in the cluster center, otherwise return to 2.
- 5) Report cluster centers

This procedure exhibits several levels of parallelism which can be exploited via the CUDA framework. Most apparent is the task parallelism between clusters. Since Equations 3 and 4 are completely independent between clusters, each iteration can be performed in c parallel tasks, one for each cluster. CUDA supports task level parallelism through the use of multiple thread blocks which, although lacking global synchronization, are effective at computing independent tasks. Within the computation on a given cluster, data parallelism is exhibited by the independent computation of membership values for each event. Since flow cytometry has a minimum of tens of thousands of events, a tremendous degree of parallelism is available. The cluster position calculation defined in Equation 7 does require global synchronization and results collection however. Pseudo code for the parallel implementation is provided in algorithm presented in Figure 2.

Input:

Events: array of event vectors

Clusters: array of current cluster centers

Output:

newClusters: array of new cluster centers

```

numerators = denominators = 0;
__syncThreads();
for (j ← 0 to n_events + n_threads){
  if (j + threadIdx.y < n_events){
    membershipValueFunc(j + threadIdx.y, blockIdx.x);
    calculateNumerator(memVal);
    incrementLocalDenominator(memVal);
  }
}
__syncThreads();
sumLocalNumerators();
sumLocalDenominators();
setNewClusters(numerators, denominators);

```

Fig. 2. Parallel C-means Iteration

C. Minimum Description Length

While the fuzzy c-means algorithm addresses some of the particular limitations of k-means, the requirement of choosing the number of clusters a priori continues to be problematic due to over-fitting and duplicate clusters. To solve this problem, the Minimum Description Length principle is applied to the final result to identify the optimal number of clusters [17].

The Minimum Description Length (MDL) principle is a formalization of Occam's Razor. The idea behind MDL is that there is a best hypothesis for any set of data that will lead to the largest compression of the data. In other words, the data can be described by using fewer symbols than are needed to describe the data literally. In this problem, this asserts that there is some optimal number of clusters than can be used to describe the data while avoiding over-fitting. Given the general nature of this assertion, many MDL formulations are possible, however the method proposed by [18] for determining the optimal number of radial basis vectors in RBF networks has been show to be effective in a fuzzy clustering environment in [2].

While some specifics of the formulation will be abstracted in this description (see [2] for full details), the essential function is to find which of the clusters produced by c-means should be included and which should be removed when determining the final cluster configuration to describe the data. The intrinsic worth of each cluster is related to the number of member events and the error introduced by describing each of those events by the single cluster center. With that must be balanced the number of member events that are also a members of other clusters. This balance can be formalized as a symmetric cost/benefit matrix, Q , where the diagonal terms q_{ii} represent the tradeoff for the i^{th} cluster and off-diagonal terms, q_{ij} , represent the crossover between clusters. The values are determined as follows:

$$q_{ii} = K_1 n_i - K_2 \xi_i - K_3 N_i \quad (8)$$

$$q_{ij} = \frac{-K_1 n_{ij} + K_2 \xi_{ij}}{2}, i \neq j \quad (9)$$

Where K_1 , K_2 , and K_3 are parameters that affect the costs of describing data, explaining error, and describing clusters respectively. The relative values of these parameters effects the scores in Q , and [18] explains how to set them. n_i is the number of events whose membership in cluster i exceed the threshold and n_{ij} is the number of events meeting this criteria for both clusters i and j . N_i is the dimensionality of the data and clusters. ξ_i and ξ_{ij} represent the error in one cluster and the overlap of two clusters respectively and are calculated by Equations 10 and 11. Let δ denote the distance function.

$$\xi_i = \sum_{x \in R_i} \delta(x, c_i) u_{xi}^p \quad (10)$$

$$\xi_{ij} = \max \left[\sum_{x \in R_i \cap R_j} \delta(x, c_i) u_{xi}^p, \sum_{x \in R_i \cap R_j} \delta(x, c_j) u_{xj}^p \right] \quad (11)$$

Here R_i is defined as the region of cluster i , or the set of all events that meet the membership criteria for that cluster. u_{xi} is the membership value, which is calculated using Equation 12.

$$u_{xi} = \frac{1}{\sum_{j=1}^c \left[\frac{\delta(x, c_i)}{\delta(x, c_j)} \right]^{\frac{2}{p-1}}} \quad (12)$$

where δ is the distance function.

The construction of the Q matrix is quite compute intensive and therefore another good candidate for GPGPU acceleration. Each of the elements is completely independent and can be assigned to different thread blocks. Within a thread block, similar methods are used to build temporary results and concatenate them together when computing ξ and n_i as were used for computing the new cluster centers. The pseudo-code is shown in the algorithm presented in Figure 3.

Input:

Events: array of event vectors
 Clusters: array of current cluster centers
 Cluster Index (i): cluster to examine

Output:

ξ_i : error associate with cluster i
 n_i : membership count in cluster i

```

localError = 0;
localMemberCount = 0;
for (j=0 to n_events + n_threads) {
  if (j + threadIdx.y < n_events) {
    membershipValueFunc(i, j + threadIdx.x);
    incLocalError(memVal2 * distance2);
    incLocalMemberCount();
  }
}
__syncThreads();
sumLocalErrors();
sumLocalMemCounts();

```

Fig. 3. Parallel ξ_i and n_i Calculation

Once the Q matrix has been constructed, a global Tabu Search method is then applied to solve for the optimal configuration of clusters to include. This is done by solving Equation 13 where h is a binary array with length equal to the number of clusters. By evaluating varying configurations defined by turning on and off clusters with h , a maximum score can be found. Tabu search works by performing modifications from a starting configuration, in this case inclusion of all clusters, iteratively. To prevent cycles, once a particular *move* is performed, the reverse move is made *tabu* for some number of iterations, referred to as the tenure. In this case, a move is the inclusion or exclusion of a particular cluster and the tabu tenure ensures the the search branches such that the subspace resulting from a particular cluster exclusion can be searched for a few iterations. Each iteration, the best available move will be selected and advance to the next iteration and the

reverse move will be made taboo. One of the benefits of tabu search over other local search strategies is that worse solutions can be accepted at any iteration, but a record is kept of the best solution found. By accepting worse solutions, it becomes possible to escape local maxima and potentially find a global maximum.

$$Score = h^T Q h \quad (13)$$

VI. TOOL IMPLEMENTATION

A computing portal can help flow cytometry, as it is fundamentally about collaboration among scientists and infrastructure experts and a large part of that is making computational resources available to scientists in a manner than they can use without concern for the underlying implementation. In flow cytometry data analysis, the scientist would like to simply supply an FCS file, possibly specify some parameters, and retrieve the results. To accomplish this, a tool chain was created that allows the scientist to set any of the internal parameters in the C-means/MDL/Tabu-Search implementation detailed above, specify some running conditions, and import either an FCS binary or a previously converted tabular text file. The user interface was implemented as a Java GUI and the selected configuration is sent as command line arguments to a perl script that launches a FCS conversion script through the statistical software package, R, if required, invokes a bash shell script to populate a header file with the selected parameters, compiles the application, and manages execution with varying numbers of clusters if requested. Figure 4 shows the interface as it is presented to the user.

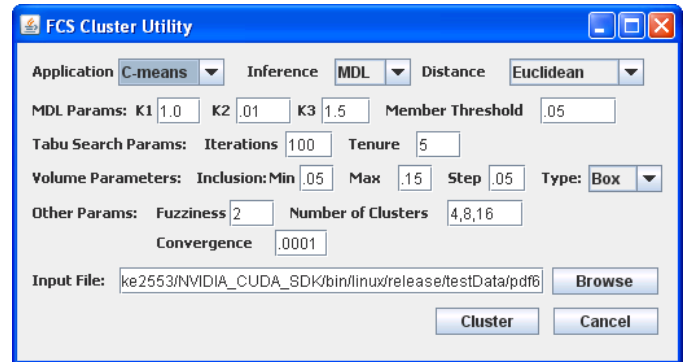


Fig. 4. FCScluster User Interface

In addition to the implemented flow based on C-means, the interface was constructed in a modular fashion such that other clustering algorithms, inference methods, distance functions, volume formulations, etc can be added easily. Since the research effort towards automated intelligent clustering of flow cytometry data is only just beginning, having an extendible framework for comparison of results is very useful.

VII. RESULTS

Two approaches to the problem of clustering high dimensional data have been presented, one based on a fuzzified K-medoids and BIC and another based on C-means and MDL.

The team members implemented these solutions completely independently and at this time there exists some discrepancy between the two implementations both with quality of results produced and effective utilization of the GPGPU architecture. With that noted, the results of each approach will be reported and a comparative discussion will follow.

A. *K-medoids and BIC*

This section will compare the performance of a sequential k-medoids algorithm against the CUDA version of the algorithm. Each version of the algorithm was tested using a 777562 by 12 FCS file. Each version was run twenty times using 2, 4, 8, 16, and 32 clusters. This was done to see how well they would perform when increasing the number of clusters. Table 1 is a summary of the results, the time was recorded in milliseconds.

	Sequential (ms)	CUDA (ms)	Speed Up
2	471	39.53	13.18
4	1199.5	44.67	27.77
8	3559.5	72.06	52.72
16	11772	140.47	98.14
32	42616	313.8	159.68

TABLE I
K-MEDOIDS PERFORMANCE SUMMARY

As you can see from Table I the performance of the sequential version became worse as the number of clusters was increased. The same is true for the CUDA version, however the CUDA version was still able to cluster the data much faster than the sequential version. Figures 5 and 6 are graphs of the performance of the CUDA and sequential versions of k-medoids.

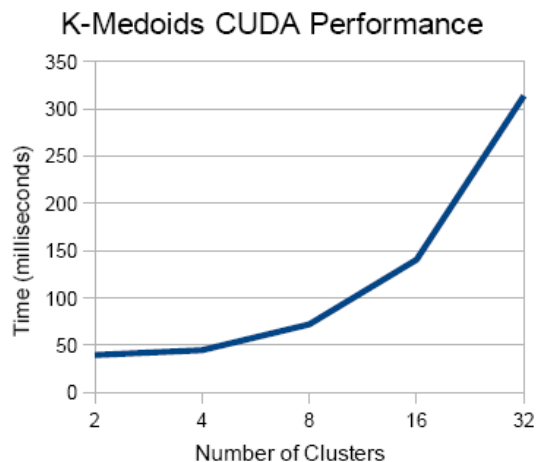


Fig. 5. CUDA Performance

Figures 5 and 6 show the execution time for the CUDA-accelerated and CPU-only implementations respectively as the number of clusters increases. Clearly the CUDA version performs in less overall time.

B. *C-means and MDL*

The object of this approach is two-fold. First, the results must show functionality and demonstrate promise for FCS

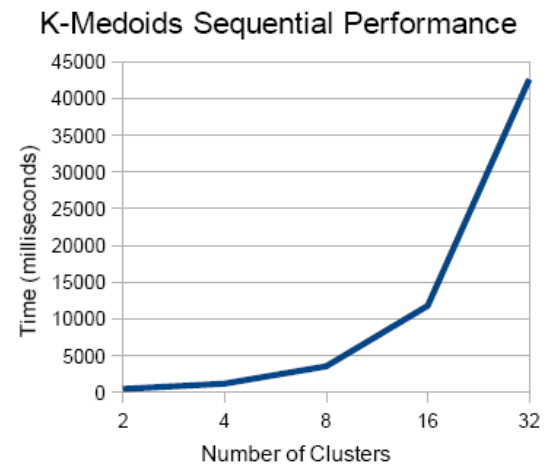


Fig. 6. Sequential Performance

clustering. To investigate this, several test data sets were generated with known clusters using the *mvtnorm* library for *R*. Functionality was shown by selecting the known cluster centers even when originally looking for more clusters than logically exist. After C-means converges on a set of cluster centers and the MDL Q matrix is generated, the Tabu Search takes over and identifies which cluster to include and which to ignore. The correct number of clusters and cluster centers could always be identified, however the MDL parameters turned out to be quite sensitive and needed to be tuned depending on the number of starting clusters selected.

The second objective is to achieve performance improvements that realize the potential of the CUDA framework and Tesla Architecture. As detailed in the preceding section, multiple levels of parallelism exist in the application and were exploited in the implementation. To gather performance data, a single data set size of 100,000 elements was used. With this held constant, the number of clusters and the number of dimensions were varied. The essential trends have been condensed into the following Figures, and some tabular results are included at the close of this section.

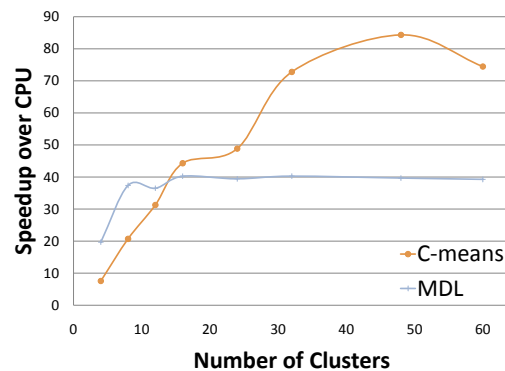


Fig. 7. C-Means Speedup vs. Clusters

As is readily apparent from Figures 7, 8, and 9, the CUDA enabled GPU version far outperformed the sequential CPU version. As the amount of work increases with the number

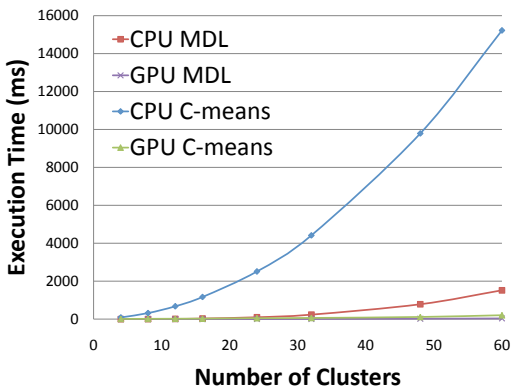


Fig. 8. C-Means Execution Time

of clusters, the CPU experiences a swift increase in execution time while the GPU retains a low rate of increase. This results because any increase in work can be executed in parallel with other work on the GPU, but the CPU requires directly increased execution time to complete the work. The C-means problem is $O(NC^2)$ where N is the number of events and C is the number of clusters. The GPU escapes this quadratic increase by exploiting the increased parallelism that results from increased clusters and only increasing the amount of work done in a thread block linearly with increasing numbers of clusters. MDL is even more dramatic, as it is $O(NC^3)$. The faster sloping increase in the MDL execution time demonstrates this and since the GPU again only increases the work in a thread block linearly, much of the remaining quadratic increase can be absorbed through parallel computation.

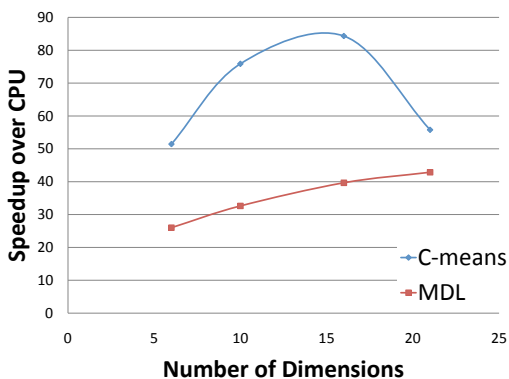


Fig. 9. C-Means Speedup vs. Dimensions

Figure 9 shows the speedup change as the number of dimensions change while holding the number of clusters constant at forty-eight. This shows an interesting result for both C-means and MDL. C-means peaks at 16 dimensions and then falls slightly (although maintaining a significant speedup). This occurs primarily because the number of concurrent threads had to be decreased to accommodate the larger memory requirements that come with additional dimensions. MDL continues to slightly increase as the memory demands are less and the same type of execution time reduction through parallelism that drove speedups in Figure 7 continue to be

beneficial. The following Tables show the raw execution time data and speedup results. The highest observed speedup are 84.34 times for each C-means iteration and 43.4 time for MDL Q Matrix Generation.

VIII. CONCLUSIONS

The performance results demonstrated from the two approaches explained in this paper show excellent speedup and make effective use of the massively parallel Tesla architecture using the CUDA framework. Further work is required to investigate data quality and intelligently move forward with improvements. The availability of such algorithms will revolutionize how flow cytometry data is analyzed. We will further optimize our algorithms to achieve even better performance and investigate other clustering techniques.

REFERENCES

- [1] Cuda zone, NVIDIA Corp. [Online]. Available: www.nvidia.com/cuda
- [2] A. Selb, H. Bischof, and A. Leonardis, "Fuzzy c-means in an mdl-framework," in *15th International Conference on Pattern Recognition (ICPR'00)*, vol. 2, 2000, p. 2740.
- [3] D. R. Parks, M. Roederer, and W. A. Moore, "A new logic display method avoids deceptive effects of logarithmic scaling for low signals and compensated data," *International Society for Analytical Cytology*, vol. Cytometry Part A 69A:, pp. 541–551, 2006.
- [4] K. M. Abbas, Y. Lee, H. Wu, and G. von Laszewski, "e-science environment for objective analysis of flow cytometry data," *gregors Flow Cytometry Paper*.
- [5] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *3rd Annual Austin Conference on Integrated Systems & Circuits 2008*, 2008.
- [6] H.-Y. Schivea, C.-H. Chiena, S.-K. Wonga, Y.-C. Tsaia, and T. Chiueha, "Graphic-card cluster for astrophysics (gracca)," in *AstroGPU*, 2007.
- [7] M. K. J. Tolke, "Towards three-dimensional teraflop cfd computing on a desktop pc using graphics hardware," Feb 2008, institute for Computational Modeling in Civil Engineering, TU Braunschweig.
- [8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, March-April 2008.
- [9] Nvidia cuda programming guide 2.0. Nvidia Corp.
- [10] K. Gulati and S. P. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Design Automation Conference*, 2008.
- [11] A. P. Reynolds, G. Richards, and V. J. Rayward-Smith, "The application of k-medoids and pam to the clustering of rules," in *Intelligent Data Engineering and Automated Learning*, ser. Lecture Notes in Computer Science. Springer Berlin, 2004, pp. 173–178.
- [12] G. Schwarz, "Estimating the dimension of a model," *The Annals of Statistics*, vol. 6, pp. 461–464, 1978, bayesian Information Criterion (BIC).
- [13] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1. Berkeley, University of California Press, 1967, pp. 281–297.
- [14] J. C. Dunn, "A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters," *Journal of Cybernetics*, vol. 3, pp. 32–57, 1973.
- [15] J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*. Plenum Press, New York, 1981.
- [16] G. Gan, C. Ma, and J. Wu, *Data Clustering Theory, Algorithms, and Applications*, M. T. Wells, Ed. Society for Industrial and Applied Mathematics, 2007.
- [17] P. D. Grunwald, *The Minimum Description Length Principle*. The MIT Press, 2007.
- [18] A. Leonardis and H. Bischof, "An efficient mdl-based construction of rbf networks," *Neural Networks*, vol. 11, issue 5, pp. 963–973, 1998.

TABLE II

EXECUTION TIME AND PERFORMANCE DATA FOR 16 DIMENSIONAL DATA

Clusters	Single CPU (ms)		GPU (ms)		Speedup	
	cmeans	MDL	cmeans	MDL	Cmeans	MDL
4	90	0.51	11.88	0.026	7.576	19.692
8	318	3.80	15.35	0.102	20.717	37.383
12	676	12.79	21.61	0.350	31.283	36.502
16	1168	30.56	26.36	0.759	44.307	40.261
24	2512	99.37	51.40	2.519	48.875	39.445
32	4418	235.14	60.69	5.837	72.794	40.287
48	9792	785.57	116.09	19.800	84.346	39.675
60	15222	1519.10	204.36	38.677	74.485	39.276

TABLE III

EXECUTION TIME AND PERFORMANCE DATA FOR 21 DIMENSIONAL DATA

Clusters	Single CPU (ms)		GPU (ms)		Speedup	
	cmeans	MDL	cmeans	MDL	Cmeans	MDL
4	95	0.64	16.947	0.039	5.606	16.449
8	330	4.76	21.574	0.159	15.297	29.988
12	672	16.25	29.440	0.451	22.826	35.996
16	1138	38.13	36.868	0.926	30.867	41.192
24	2464	123.71	72.708	2.940	33.889	42.080
32	4302	293.19	86.662	6.755	49.641	43.401
48	9428	987.33	168.959	23.022	55.801	42.886
60	14774	1930.56	267.269	44.894	55.278	43.003