Scalable Data Clustering using GPU Clusters

Andrew Pangborn, Gregor von Laszewski, James Cavenaugh, Muhammad Shaaban, Roy Melton

CONTENTS

Scalable Data Clustering using GPU Clusters

Andrew Pangborn, Gregor von Laszewski, James Cavenaugh, Muhammad Shaaban, Roy Melton

***Abstract*—The computational demands of multivariate clustering grow rapidly, and therefore processing large data sets, like those found in flow cytometry data, is very time consuming on a single CPU. Fortunately these techniques lend themselves naturally to large scale parallel processing. To address the computational demands, graphics processing units, specifically NVIDIA's CUDA framework and Tesla architecture, were investigated as a low-cost, high performance solution to a number of clustering algorithms.**

C-means and Expectation Maximization with Gaussian mixture models were implemented using the CUDA framework. The algorithm implementations use a hybrid of CUDA, OpenMP, and MPI to scale to many GPUs on multiple nodes in a high performance computing environment. This framework is envisioned as part of a larger cloud-based workflow service where biologists can apply multiple algorithms and parameter sweeps to their data sets and quickly receive a thorough set of results that can be further analyzed by experts.

Improvements over previous GPU-accelerated implementations range from 1.42x to 21x for C-means and 3.72x to 5.65x for the Gaussian mixture model on non-trivial data sets. Using a single NVIDIA GTX 260 speedups are on average 90x for C-means and 74x for Gaussians with flow cytometry files compared to optimized C code running on a single core of a modern Intel CPU. Using the TeraGrid "Lincoln" high performance cluster at NCSA C-means achieves 42% parallel efficiency and a CPU speedup of 4794x with 128 Tesla C1060 GPUs. The Gaussian mixture model achieves 72% parallel efficiency and a CPU speedup of 6286x.

I. Introduction

Science and business applications often produce massive data sets. This immense amount of data must be classified into meaningful subsets for data analysts and scientists to draw meaningful conclusions. Data clustering is the broad field of statistical analysis that groups similar objects into relatively homogenous sets called clusters. Data clustering has a history in a wide variety of fields, such as data mining, machine learning, geology, astronomy, and bioinformatics, to name a few [1] [2]. The nature of the data similarity varies significantly from one application and data set to another. Therefore no single data clustering algorithm is superior to all others in every instance. As such, there has been extensive research and a myriad of clustering techniques developed in the past 50 to 60 years [2].

Flow cytometry is a mainstay technology used in immunology and other clinical and biological research areas such as DNA analysis, genotyping, phenotyping, and cell function analysis. It is used to gather information about the physical and chemical characteristics of a population of cells. Flow cytometers produce a d-length multidimensional data vector of floating point values for every event (usually a cell) in a sample, where d indicates the number of photo-sensitive sensors installed. Typical samples have on the order of 10^6 events with upwards of 24 dimensions (and this number is expected to continue increasing as flow cytometer technology improves). This massive amount of data must then be clustered in order for biologists to draw meaningful conclusions about the characteristics of the sample.

Sequential bivariate gating is the approach traditionally followed by biologists for clustering and analyzing flow cytometry data. Two dimensions of the data are analyzed at a time with a scatter plot. Clusters are then manually drawn around populations of cells by a technique called gating. The data sets are typically diffuse and clusters are not always well-defined and distinct; therefore gating requires experience and expert knowledge about the data and the dimensions involved. Unfortunately this process is time consuming, cumbersome, and in-exact. Unpublished research by the University of Rochester Center for Vaccine Biology and Immunology suggests that results can vary by as much as an order of magnitude between experienced immunologists on a difficult data set. Therefore both the number and quality of the analyses produced by sequential bivariate gating is limited. It is also impractical to analyze many samples individually in a large scale patient trial with manual gating.

Multivariate data clustering techniques have been around for decades; however their application to the field of flow cytometry has been limited. There has been a recent surge in research activity over the past few years applying multivariate data clustering to flow cytometry data. Multivariate techniques have the potential to use the full multidimensional nature of the data, to find cell populations of interest (that are difficult to isolate with sequential bivariate gating), and to allow analysts to make more sound statistical inferences from the results. Flow cytometry data sets are complex, containing millions of events, dozens of dimensions, and potentially hundreds of natural clusters. Unsupervised multivariate clustering techniques are computationally intensive, and the computational demands grow rapidly as the number of clusters, events, and dimensions increase. This makes it very time consuming to analyze a flow cytometry data set thoroughly using a single general purpose processor. Fortunately, many clustering techniques lend themselves nicely to large scale parallel processing.

In this paper NVIDIA's CUDA framework for general purpose computing on graphics processing units (GPGPU) was investigated as a low cost, high performance solution to address the computational demands of unsupervised multivariate data clustering for flow cytometry [3]. However, the algorithms themselves are applicable to a wide range of applications, not just flow cytometry. The existing work on data clustering algorithms using GPGPU has been limited in the algorithms implemented and the scalability of such algorithms (using multiple GPUs and clusters of GPUs). Two unsupervised multivariate clustering algorithms, C-means and Expectation Maximization with Gaussian mixture models,

were implemented using CUDA and the Tesla architecture. Multiple GPUs on a single machine are leveraged using shared memory and threading with OpenMP. The parallelism is expanded to support GPUs spread across multiple nodes in a high-performance computing environment using MPI. Functionality is verified for all methods using synthetic data. Real flow cytometry data sets are used to assess the accuracy and quality of results. The performance of sequential, single GPU, and multiple GPU implementations are compared in detail.

The remainder of this paper is organized as follows. Section II provides an overview of data clustering, GPGPU, and previous work. After motivating the use of GPUs for data clustering, Section III provides implementation details for the parallel clustering algorithms and discusses improvements over previous work. Section IV analyzes results for the parallel clustering algorithms using a variety of different performance metrics. Finally, Section V concludes the paper and provides suggestions for future work.

II. Background

This section provides an overview of data clustering, modern GPU architectures, NVIDIA's Compute Unified Device Architecture (CUDA) for General Purpose Computation on Graphics Processing Engines (GPGPU), and discusses the previous work with data clustering using GPUs.

A. Data Clustering

Data clustering is a statistical method for grouping of similar objects into related or homogeneous sets, called clusters. There are a myriad of scientific fields and commercial applications that generate immense amounts of data, ranging from high energy particle physics at CERN to the buying habits of consumers at grocery stores. In any case, the objective is to group related data together so that analysts can draw meaningful conclusions from the data. The goal of data mining as well as the size and nature of the data varies tremendously from one field to another, and even from one data set to another in a given field. As such, there has been a wide variety of data clustering techniques developed over the past 60+ years.

Many popular data clustering techniques have a combinatorial increase in computational time complexity as the number of dimensions, the number of vectors, and the number of clusters increase. The nature of the data in flow cytometry, with millions of events, over 20 dimensions, and potentially hundreds of natural clusters, make unsupervised data clustering both difficult and very computationally demanding.

The implementation in this paper focuses on two popular data clustering algorithms: C-means and Expectation Maximization with a Gaussian mixture model. C-means is a center-based algorithm that attempts to minimize the squared-error between all data points and their closest center. It is a soft (fuzzy) version of the popular K-means clustering algorithm. "Soft" clustering means that every data point has a fractional membership in each cluster (where the sum of the probability is 1 and all values are in the range [0,1]), whereas "hard" clustering means that every data point belongs to a single cluster (or one can define it as a special case of soft clustering where all probabilities belong to the set 0,1). Expectation Maximization with a Gaussian mixture model works based on the principle that the data set can be approximated by a collection (mixture) of multi-dimensional Gaussian (normal) distributions. Both algorithms follow a two-step iterative approach where data point membership values are determined by the current clusters (a center location for C-means or a Gaussian distribution for the mixture model), and then the clusters are updated based on the new membership values.

B. GPGPU

Graphics processing units (GPUs) have evolved from simple fixed function co-processors in the graphics pipeline to programmable computation engines suitable for certain general purpose computing applications. The introduction of programmable shaders into GPUs made the field of general purpose computing on graphics processing units (GPGPU) possible. Older efforts at GPGPU required researchers to cast the general purpose computations into streaming graphical applications, with the instructions written as shaders, such as the OpenGL Shader Language (GLSL) and the data stored as textures.

With the Geforce 8800 graphics card series, NVIDIA introduced a new architecture with a unified shader model [4]. This architecture was a major shift from a fixed-function pipeline (with separate processing elements dedicated to particular tasks, such as vertex shading and pixel shading) to a more general purpose architecture. In graphics applications, these processing elements still execute either vertex or pixel shading procedures. However, they are actually multiprocessors capable of executing threads in general purpose kernels. A kernel in this context simply means a subroutine (function) that executes on the GPU's processors. Generally the same kernel is executed simultaneously on all processors, but each thread performs operations on different ranges of data (single-instruction multiple-data, or SIMD). This is a massively parallel architecture with many benefits over a general purpose desktop processor for data-intensive computations such as data clustering.

The NVIDIA GT200 GPU architecture contains upwards of 240 concurrent processing elements [4]. NVIDIA's latest architecture code-named Fermi (GF100) has upwards of 512 cores, supports multiple simultaneous kernels, and has a caching hierarchy for the device memory - making it even closer to a general purpose processor (but with many more cores) [5]. Compared to general purpose processors, a much larger portion of on-chip resources in a GPU is dedicated to data and floating-point calculations rather than control and sequencing — ideal for many data clustering algorithms, which are composed almost entirely of floating-point operations and have very few branches.

C. CUDA

The compute unified device architecture (CUDA) is a framework for scientific general purpose computing on NVIDIA GPUs. CUDA provides a set of APIs, a compiler (nvcc),

supporting libraries, and hardware drivers to enable running applications on the GPU. Programs utilize the CPU on the workstation, the *host* in the CUDA documentation, as well as the GPU which is the *device*. CUDA uses a superset of ANSI C with some extensions. It has additional identifiers to specify whether functions are defined for the host only, for the device only, or globally (kernels callable by the host). There are also identifiers to specify the memory location of variables in kernels (either shared or global memory). Finally there is additional syntax added for invoking kernels.

At heart of the Tesla Architecture [4] are streaming multiprocessors (SMs). Each SM is comparable to a simple CPU - it has an interface for fetching instructions, decoding instructions, functional units (an ALU), and registers. Within each SM are many functional units, called "cores" which all execute different threads with the same instruction but on different data elements (Single Instruction Multiple Data — SIMD). SMs also have a small amount (16 KB) of high-speed memory for sharing data between threads on a single SM, and an interface to the rest of the onboard DRAM. The NVIDIA GTX 260 for example has 24 multiprocessors with 8 cores each, for a total of 192 simultaneous execution cores.

Since understanding the thread model is essential to effective programming with CUDA and understanding CUDA program implementations, this section provides a brief overview. For a more detailed explanation, please consult NVIDIA's CUDA Programming Guide [6]. At the top level, threads are organized into a *grid* which composes the entirety of the application running on the GPU at any given time (i.e. a kernel launched by the host). The grid contains a 2-dimensional set of *blocks*. A block runs on a single multiprocessor and cannot be globally synchronized with other blocks and is not even guaranteed to run physically at the same time as other blocks in the grid. The number of blocks can, and typically should, exceed the number of multiprocessors on the GPU. New blocks in the grid will be allocated to multiprocessors once the previous set of blocks finishes executing.

Inside a block are threads with 3-dimensional indices and they are allocated to the different cores within a multiprocessor. The multi-dimensional indices allow the programmer more easily to map kernels to 2D or 3D problems (such as texture locations or X,Y,Z vertex coordinates in a graphics application). Regardless of the number of index dimensions in use, the maximum number of threads within a block is 512. Threads are organized into *warps*, which are sets of 32 threads executing the same instruction in an SIMD fashion. All threads within a block have their own registers and can access the shared memory on the SM. A low-overhead thread synchronization function is available for all threads within a block, and functions the same as a barrier in OpenMP and MPI applications.

Just like understanding the thread model is important for parallelizing algorithms and implementing them with CUDA, understanding the memory model is essential to high performance CUDA applications. Again this section provides a brief overview — for more details consult the CUDA programming guide [6]. Memory on CUDA-capable GPUs is divided into three main categories: registers, shared memory, and global memory. Registers are divided evenly amongst the active threads on the SM (which can be from one or more thread blocks depending on resource consumption). A thread cannot access the register of another thread. Shared memory is a small high speed memory inside each SM that is equivalent in access time to the registers if no memory bank conflicts occur. All threads within a block can access the same shared memory. Finally the global memory is the large off-chip DRAM. It has a very large latency (hundreds of cycles), but it can be accessed by all threads in the kernel and it persists throughout the lifetime of the applications (can be reused by multiple kernels). Unlike modern CPUs which have a sophisticated multi-level caching hierarchy to hide the large latency of system (global) memory, CUDA programmers have to manually place frequently accessed code into the limited (but high speed) shared memory. Programs must also ensure that memory accessed are contiguous and aligned to multiples of 16 times the size of the data element in order to maximize memory throughput when accessing global memory [7].

D. Previous Work

The abundant parallelism and large number of floating point operations make data clustering algorithms a natural choice for implementation using GPGPU. In 2004, Hall et al. implemented K-means using *Cg* and achieved a speedup of 3x versus a modern cpu at the time the article was written. In 2006, Takizawa et al. implemented K-means using fragment shaders and NVIDIA 6800 GPUs [8]. The implementation in [8] only showed a speedup of 4x relative to a cluster of CPUs without GPUs; however their implementation divided the task among a cluster of PCs each equipped with GPUs using MPI. These efforts showed it was possible to implement a data clustering algorithm using a graphics pipeline and achieve speedup and to distribute that work at a coarse-grained level to multiple GPU co-processors.

The introduction of more advanced GPU architectures and coding frameworks for general purpose computing on GPUs allowed for much more significant speedup of data clustering algorithms on GPUs. Che et al. implemented K-means with an impressive speedup of 72x using CUDA and a Nvidia 8800 GTX GPU in 2008 [9], and also compared it to a multi-threaded version running on a Quad core processor, and still maintaining a speedup of 30x.

While the performance results of recent K-means implementations on GPUs and other parallel architectures are impressive, K-means is an embarrassingly parallel algorithm and its spherical bias is not very good at analyzing flow cytometry data, where clusters often have very diverse non-spherical shapes. Outliers can also have a significant impact on the resulting cluster centers. Despite these short-comings, K-means is still a de facto standard clustering algorithm used in a variety of applications, which has been implemented on many platforms and parallel architectures, and thus is a good basis for comparison.

Using a fuzzy version of K-means, where data points have a membership value in all of the clusters, rather than belonging to only one cluster, can lessen the effect of outliers. It is also

produces better results when the number of specified clusters does not match the number of natural clusters in the data. A hard clustering may attempt to create multiple adjacent, but not overlapping, clusters inside one natural cluster. A soft clustering is more likely to have multiple overlapping clusters with approximately the same center — which more accurately reflects the underlying data. Therefore the paper will implement and examine a C-means (the literature uses C for soft clustering, and K for hard clustering) algorithm.

Anderson et al. implemented C-means using Cg (C for graphics) with two non-Euclidean distance measures in 2007 with a maximum speedup of 97x [10]. The non-Euclidean distance measures involved a covariance matrix, but they limited it to diagonal covariance. Anderson et al. published another paper in August 2008 on c-means using the standard Euclidean distance with speedups of 107x on the 8800 GTX [11]. Shalom et al. implemented C-means on an NVIDIA 8800 GTX using the OpenGL Shader Language with speedup of 94x [12]. The Shalom et al. implementation focuses on the ability to scale to an arbitrarily large number of dimensions and clusters. This results in a significant amount of CPU to GPU data transfer, limiting the performance of the algorithm. In 2009 Espenshade and Pangborn et al. implemented C-means with an MDL information criterion using CUDA on a single GPU. The implementation achieved speedup of over 70x on flow cytometry data compared to a naive C implementation [13]. This paper significantly improved that implementation, increasing performance and scalability significantly on a single GPU and extending it to multiple GPUs.

It is often difficult to compare results directly to previous GPU acceleration work because the details of the experiment are not clear. Authors post speedup ratings but they depend largely on the quality of the CPU reference version. Absolute execution times are a more realistic metric for comparing multiple implementations, but not all of these papers provide that information. Additionally, some of the ones that do provide execution times such as [12] do not state the number of iterations; the execution time depends not only on the convergence criterion and data dimensions, but also the actual values of the data in the experiment.

In addition, this paper implemented an Expectation Maximization (EM) algorithm with Gaussian Mixture Models (GMMs). A publication from Kumar et al. [14] implemented EM with GMMs using CUDA. Using hardware similar to the aforementioned CUDA implementations of C-means, it achieved a speedup of 120x for particular data sizes. One limitation of this implementation is that it uses only diagonal covariance matrices, rather than the full covariance matrices for the Gaussian Mixture Models. This reduces the complexity significantly; however it does not allow for dimensions to be statistically dependent upon each other — which often occurs in real data sets. It also does not make use of multiple GPUs nor include any information criterion for unsupervised assessment of clustering results. Another significant disadvantage of the Kumar et. al implementation is that it requires a very large amount of memory for the M-step computation. It requires an $M \times ND$ matrix to perform the covariance kernel. Thus if there are D = 24 dimensions, N = 500,000 events, and M = 100 clusters (reasonable numbers for a single flow cytometry datafile), it would require almost 5 GB of memory — an amount that even the current high-end Tesla cards do not have. A few other CUDA applications have been developed using GMMs, such as anomaly detection in hyperspectral image processing [15] and have achieved overall speedup factors of 20x, and over 100x for specific portions of the algorithm.

In 2010, M. A. Suchard et al. [16] used multiple GPUs and CUDA for Markov Chain Monte Carlo (MCMC) and Bayesian mixture models with flow cytometry data. Our work, developed independently and in parallel to Suchard et al. uses multiple GPUs combined CUDA, OpenMP and MPI for large scale data clustering using C-means and Expectation Maximization with Gaussian mixture models.

III. Clustering Implementations

This section discusses the parallel implementations of the clustering algorithms. Both algorithm implementations utilize NVIDIA's CUDA driver and software development environment for GPGPU. The overall program flow and some kernel details are discussed for each algorithm. Pseudocode is presented for some of the CUDA kernels. The pseudocode listings for kernels are generally detailed and based directly on the actual implementation — only a few details are occasionally omitted such as variable declarations, memory copying, memory management, constant definitions, and boundary conditions for the sake of brevity and readability. The full source code is available online, please refer to the Appendix for more details.

Throughout the remainder of the article: N represents the number of events (data vectors), D is the number of dimensions for each event, M is the number of clusters, and G is the number of GPUs.

A. Parallelism Strategy

Each algorithm is designed to be scalable to one or more GPUs. Due to how the NVIDIA runtime API operates, each CUDA device requires its own host thread. The typical approach to multithreading in C/C++ is an OS specific multithreading approach such as pthreads or Windows threads. These approaches are not very portable, and their APIs typically obfuscate the underlying algorithm.

The OpenMP library for shared-memory parallel computing was chosen as the platform for managing threads and communicating between multiple GPUs on a single node. The programming model is simplistic and consistent with coding in CUDA kernels, which also uses shared memory and thread barriers for synchronization. OpenMP is supported by GCC 4.2+ and Microsoft's Visual C++ compiler without the need to install any additional software.

Communicating between multiple physical nodes in a distributed memory environment cannot be readily achieved using a shared memory library such as OpenMP. Therefore the cluster implementations of the algorithms use a hybrid of MPI for communication between nodes, and OpenMP for multiple threads within each node.

The specifics of the two clustering algorithms differ, but the overall structure of how work is distributed to different nodes, processors, and GPUs is the same. A hybrid of MPI, OpenMP, and CUDA is used. MPI is a message passing interface that allows communication between different processes in a distributed memory environment, and thus it maps well to the different physical machines (nodes) in a computing cluster. OpenMP allows multiple threads to communicate via shared memory and maps well to multiple CPUs in a single node and multi-core CPU environments. Finally, GPU co-processors with the CUDA toolkit and driver accelerate the calculations done by each thread.

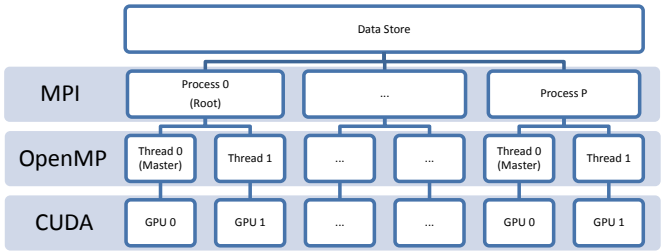

Fig. 1. Process Hierarchy

Figure 1 shows the structure of the hybrid parallel environment. At the top of the hierarchy, MPI launches P processes — one for each machine (node) in the cluster. Each MPI process connects to a data store to load the input data. In the experiments executed in this paper each machine loads data from a file, but there is nothing preventing it from extracting data from a database, data grid, or other storage mechanism. The workload is mapped evenly to the different processes. Each process then enters a parallel OpenMP code section with multiple threads. The number of threads is equal to the number of CUDA capable GPUs found on the machine. Each OpenMP thread selects a GPU based on its thread ID and creates a CUDA context. Just as the data set gets split among the MPI processes, the data set for the node is split between the thread/GPU pairs.

Figure 2 shows the multi-level MapReduce structure used for communication between the processes and threads shown in Figure 1. The same logic applies for P processes, but is shown with only two for simplicity. The master MPI node (rank 0) acts as the root for all collective operations, such as MPI_Broadcast, MPI_Scatter, MPI_Gather, MPI_Reduce, etc. The master node is also a worker node. It uses MPI_IN_PLACE to declare that the send buffer and the recv buffer are identical — this is indicated by the dashed "implicit" communication arrows in Figure 2. Within each node, T0 is the master OpenMP thread. Only the master threads make MPI calls and perform reduction.

In summary, a three-tiered MapReduce parallel structure is used. The first two levels use MPI and OpenMP to handle coarse grained parallelism by mapping the data set to individual nodes and threads. Each thread uses CUDA and a GPU co-processor for all of the computationally intensive tasks. The CUDA architecture itself is multi-tiered with coarse-grained data independent blocks and fine-grained groups of SIMD threads. Finally results are reduced by the master OpenMP threads and MPI root node and are broadcast to all workers for the next iteration.

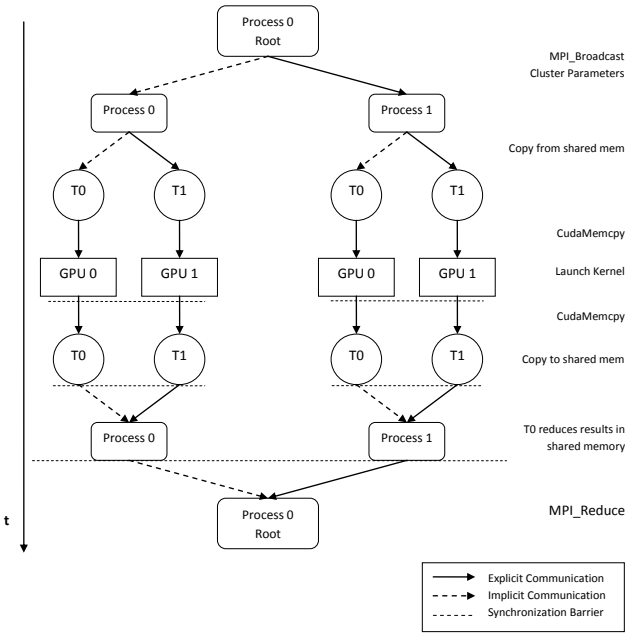

Fig. 2. Communication and Reduction Hierarchy

B. C-means

The standard C-means implementation iteratively optimizes the cluster centers by updating the cluster memberships with equation (1) and then computing new cluster centers using equation (2). The two-step iteration continues until the cluster centers converge (the change in position is less than the user-specified tolerance) to a local minima.

$$u_{ij} = \frac{1}{\sum_{m=1}^{M} \left(\frac{\|x_i - c_j\|}{\|x_i - c_m\|} \right)^{\frac{2}{p-1}}} \tag{1}$$

$$c_j = \frac{\sum_{i=1}^{N} u_{ij}^{p} * x_i}{\sum_{i=1}^{N} u_{ij}^{p}} \tag{2}$$

Naively parallelizing the computation by unrolling the loops into SIMD operations and distributing the computations to different thread blocks and threads is not very efficient since the equations have a lot of redundant calculations. For example, the denominator of the summation in Equation 1 is the same for all clusters. This can be improved by transforming Equation 1 to Equation 3. Other optimizations include caching the distances between data vectors and each cluster center before computing the new membership values.

$$s_i = \sum_{k=1}^{M} \left(\frac{1}{\|x_i - c_k\|^{\frac{2}{p-1}}} \right) \quad \forall\, i \in [1, N]$$

$$u_{ij} = \frac{1}{(\|x_i - c_j\|)^{\frac{2}{p-1}} * s_i} \quad \forall\, i \in [1, N], \forall\, j \in [1, M] \qquad (3)$$

The C-means implementation in this article uses a total of four CUDA kernels — *DistanceMatrix*, *MembershipMatrix*, *UpdateCenters*, and *ClusterSizes*. The following list outlines the steps of the full algorithm.

1) Root node reads input data from file and then scatters data to other nodes
2) Seed cluster centers with random data points
3) Copy input data to GPU
4) Copy cluster centers to GPU
5) DistanceMatrix kernel
6) MembershipMatrix kernel
7) UpdateCenters kernel, copy partial centers to host from each GPU
8) ClusterSizes kernel, copy cluster sizes to host from each GPU
9) Aggregate partial cluster centers and reduce
10) Compute difference between current cluster centers and previous iteration. If greater than epsilon, return to step 4.
11) Compute cluster distances and memberships using final centers
12) Gather membership values from all nodes
13) Output final clustering results

DistanceMatrix computes a $M \times N$ matrix that contains the Euclidean distance from each data point to every cluster center using a $M \times \lceil N/512 \rceil$ kernel grid — one thread per distance calculation. The *MembershipMatrix* converts distances into membership values for each cluster. The kernel uses a $\lceil N/512 \rceil$ grid of thread blocks, where each thread computes M membership values for a data vector using Equation 3. The first half of Equation 3, s_i, is computed once and then used for M membership calculations.

UpdateCenters computes the numerator of Equation 2 using a $D \times \lceil M/4 \rceil$ kernel grid. Although a $D \times M$ grid exposes more parallelism, it generates many more memory requests than a single-threaded sequential implementation needs. The event data are the same for every cluster. Computing the M clusters independently means that the event data gets iterated over M times from global memory. Similarly, the membership values get iterated over D times for each cluster center. Unfortunately the limited size of shared memory on the GPU and the very large quantities of data involved do not permit the working set to be in fast memory. The final kernel implementation attempts to strike a balance between having sufficient parallelism to use all of the GPU resources and reducing the number of memory accesses. A grid of $D \times M/B$ blocks is used, where B indicates the number of cluster centers computed per block. This grid reduces the number of times the input data are loaded from global memory by a factor of B. Partial results are required for each cluster in a block; therefore shared memory usage increases proportional to B. A value of $B = 4$ allows the kernel to maintain 50%+ occupancy on all devices and performs approximately 30% faster than the $M \times D$ grid on a typical flow cytometry file.

Finally the *Sizes* kernel computes the denominator of Equation 2 using a grid of M blocks (one per cluster) with 512 threads per block. The *host* then collects the numerators and denominators from every GPU and computes the final cluster center values using Equation 4.

$$c_j = \frac{\sum_{g=1}^{G} numerator_{jg}}{\sum_{g=1}^{G} denominator_{jg}} \quad \forall \;\; j \in [1, M] \qquad (4)$$

C. Expectation Maximization with Gaussian mixture models

Data in flow cytometry and many other fields is composed of many possibly overlapping clusters. The mixture model approach suggests that the data set is an aggregate of (or at least can be approximated by) a mixture of multiple distinct behaviors. Gaussians mixtures form probabilistic models composed of multiple distinct Gaussians distributions as clusters. Given a D dimensional data set, each cluster m is characterized by the following parameters [17].

N_m : the number of samples in the cluster
π_m : the probability that a sample in the data set belongs to the cluster
μ_m : a D dimensional mean
R_m : a $D \times D$ spectral covariance matrix

Assuming there are N data points $y_1, y_2, \cdots, y_N$, then the probability that an event y_i belongs to a Gaussian distribution is given by the following Equation [17].

$$p(y_n|m,\theta) = \frac{1}{(2\pi)^{D/2}\,|R_m|^{1/2}} \exp\left\{ -\frac{1}{2}(y_n - \mu_m)^t R_m^{-1} (y_n - \mu_m) \right\} \qquad (5)$$

Neither the statistical parameters of the Gaussian Mixture Model, $\theta = (\pi, \mu, R)$, nor the membership of events to clusters are known a priori. An algorithm must be employed to deal with this lack of information. Expectation maximization is a statistical method for performing likelihood estimation with incomplete data [2]. The objective of the algorithm is to estimate θ, the parameters for each cluster.

First each event y_n is classified based on the likelihood criteria in Equation (5). This step is the E-step of the EM algorithm. Instead of a hard classification based on the maximum likelihood, it is desirable to compute a soft classification (membership value) for each event and each cluster. The membership value is the ratio of the weighted likelihood to the total weighted likelihood of all clusters — see Equation (6).

$$p(m|y_n,\theta) = \frac{p_{y_n|x_n}(y_n|m,\theta)\pi_m}{\sum_{l=1}^{M} p(y_n|l,\theta)\pi_l} \qquad (6)$$

The actual computation uses the log-likelihood to prevent overflow and underflow. The first step computes log-likelihood via Equation 7, and then a log sum of exponentials (Equation 8) is used for the denominator of the membership values

(Equation 6). In order to avoid potential overflow of the exponential function, the equality in Equation (8) is used. The maximum value computed by the exponential function will be zero (resulting in 1.0) and small values will approach zero.

$$\frac{1}{(2\pi)^{M/2} det(R_m)^{1/2}} - \frac{1}{2}(y_i - \mu_m)^T R_m^{-1}(y_i - \mu_m) \quad (7)$$

$$\log\left(\sum_i \exp(x_i)\right) \equiv \max(x) + \log\left(\sum_i \exp(x_i - \max(x))\right) \quad (8)$$

The cluster parameters, θ, are re-estimated based upon the new membership values completed in the E-step using Equations 9, 10, 11, and 12 [17]. The event classification (E-step) and re-estimation of cluster parameters (M-step) repeats until the change in likelihoods for the events is less than some ϵ.

$$\bar{N}_m = \sum_{n=1}^{N} p(m|y_n, \theta) \quad (9)$$

$$\bar{\pi}_m = \frac{\bar{N}_m}{N} \quad (10)$$

$$\bar{\mu}_m = \frac{1}{\bar{N}_m} \sum_{n=1}^{N} y_n p(m|y_n, \theta) \quad (11)$$

$$\bar{R}_m = \frac{1}{\bar{N}_m} \sum_{n=1}^{N} (y_n - \bar{\mu}_m)(y_n - \bar{\mu}_m)^t p(m|y_n, \theta) \quad (12)$$

The computation uses six CUDA kernels — two for the E-step and four for the M-step. The following outlines the steps of the EM with a Gaussian mixture model algorithm.

1) Read input data from file.
2) Copy input data to GPU.
3) Initialize Gaussian model parameters.
4) Copy model parameters to GPU.
5) Launch E-step kernels, aggregate likelihood value from each GPU.
6) Launch M-step kernels, aggregate parameters from each GPU.
7) If change in likelihood greater than epsilon, return to step 5
8) Copy membership values to host.
9) Compute MDL score for current cluster configuration.
10) Combine two most similar Gaussian models.
11) If number of Gaussians is greater than target number of clusters, return to step 4.
12) Output cluster configuration with minimum MDL score.

1) E-step Kernels: The *Likelihood* kernel computes an MxN matrix of likelihood values. It is comparable to the *Distance* kernel of the C-means implementation, but calculates Equation 6 instead of Euclidean distances. The equation involves the multiplication of a shared $D \times D$ matrix with a D length vector. Since the matrix can be shared between all likelihood calculations for a particular cluster, the kernel uses a coarser grid than the *Distance* kernel. The calculation is divided into a $16 \times M$ grid. Each thread block computes $N/16$ likelihoods and threads within the block compute individual likelihoods. The value of 16 strikes a balance between providing enough blocks to keep the GPU resources occupied with a small number of clusters and still minimizing the number of times the shared resources must be read from global memory.

Following the *Likelihood* kernel is the *Membership* kernel, which transforms likelihood values into membership probabilities and sums up the total likelihood. The kernel is launched with a grid containing two thread blocks per multiprocessor. This grid size maintains full occupancy of GPU resources while minimizing the number of partial likelihood summation results returned to the CPU; each block computes the sum of its likelihoods and the host performs the final reduction.

2) M-step Kernels: The M-step computes the new Gaussian distribution parameters using Equations 9, 10, 11, and 12. Each of these equations has different degrees of parallelism and requires different data to be loaded into shared memory. Therefore each equation is separated into a separate kernel. The first kernel computes N, the size of each cluster, which is just a simple summation of all the membership probabilities for each cluster. The sizes are computed using a grid of M thread blocks — one per cluster. The 256 threads within the block each add up $N/G/256$ membership values and then the threads perform a butterfly reduction (log_2) to get the final summation. When multiple GPUs are being used simultaneously ($G > 1$), then the *host* does an additional reduction and distributes the final cluster sizes, as described in Section III-A. The second kernel computes the D-dimensional mean for every cluster. The implementation is similar to first kernel, except it is launched with a grid of $M \times D$. Each thread block computes a summation of the data weighted by the membership probabilities as seen in Equation 11.

The third kernel computes the covariance matrices for each cluster according to Equation 12. There are $D \times D$ independent matrix values that must be computed for M clusters. The kernel is launched with a $\lceil M/6 \rceil \times D(D+1)/2$ grid. Each thread block is loop unrolled and computes the matrix location for up to six clusters. The unrolling allows the data points to be loaded from memory fewer times (since it can be shared between the clusters) and attempts to strike a balance between memory consumption, device resources (shared memory and register usage), and still providing enough thread blocks to keep the card fully utilized across a wide range of input parameters. The second grid dimension, $D(D+1)/2$ corresponds to the number of components in the triangular version of a $D \times D$ matrix — covariance is symmetric.

IV. Results

Performance was analyzed using a variety of different tests and metrics. First the algorithms were analyzed for GPU resource utilization — register usage, shared memory usage, kernel occupancy, and block count. Secondly the algorithms were tested to see how execution time varies by increasing the number of clusters, the dimensionality, and the number of input vectors. Finally the algorithms were compared to CPU reference versions for speedup. The multi-GPU implementations were tested for horizontal scalability by increasing the

number of GPUs and comparing the speedup to a single GPU implementation. Both strong scaling (with a fixed problem size) and weak scaling (where the workload per node remains constant — also known as time-constrained scaling [18]) results are analyzed.

A. Testing Environment

Three testing environments were used for the work in this paper. The primary environment used for development and testing was a local server at the Rochester Institute of Technology. It contains two Intel Xeon 2.5 GHz Quad Core E5420 processors with 16GB of DDR2 and two CUDA cards — a Tesla C870 and a GTX 260. The C870 is an older 1.0 compute capability card with 16 multiprocessors (128 total cores) operating at 1.35 GHz and 1536 MB of 800 MHz GDDR3 memory with a 384-bit memory bus. The GTX 260 is a newer 1.3 compute capability card with 24 multiprocessors (192 total cores) operating at 1.30 GHz and 896 MB of memory operating at 1 GHz with a 448-bit memory bus. The use of these heterogeneous cards makes it possible to see how the algorithm performs across two different GPU architectures with different numbers of cores. The C870 has much stricter global memory coalescing rules and half the number of registers (8192 versus 16384 on the GTX 260). The development environment was installed on a 64-bit Ubuntu LTS server using GCC 4.2.4 and version 2.3 of the CUDA driver and SDK.

The second testing environment was used for scalability testing as the number of the GPUs increases. Lincoln [19] is a high performance computing cluster at the National Center for Supercomputing Applications (NCSA) at the University of Illinois Urbana-Champaign, which is part of the TeraGrid. The cluster has 192 nodes, each with two Intel 64 Harpertown 2.33 GHz quad core processors and 16GB of memory. There are 96 Tesla S1070 accelerator units, each of which contains four Tesla C1060 GPUs. This results in a total of 384 GPUs (two per server node), each with 30 multiprocessors (240 total cores) operating at 1.3 GHz and 4 GB of memory with a 512-bit 800 MHz memory bus. The development environment uses RHEL4, GCC 4.2.4, CUDA 2.3, and MVAPICH 2.1 for MPI. Cluster nodes are linked together with Infiniband SDR and use the Torque batch queueing system [19].

B. Resource Utilization

C. Single-Node

Figure 3 shows the execution time for 100 iterations of the C-means CUDA implementation on a data set with 24 dimensions and 100 clusters. The dimensions and number of clusters are held constant while the number of data points is increased. The algorithm complexity is linear with respect to N, the number of data points, and the actual performance is also approximately linear for $N > 10,000$.

Figure 4 shows speedup of the GPU-accelerated version compared to a single-core CPU-only reference version. There are four different GPU configurations - three of which are using a single GPU (Tesla C870, Tesla C0160, or Geforce

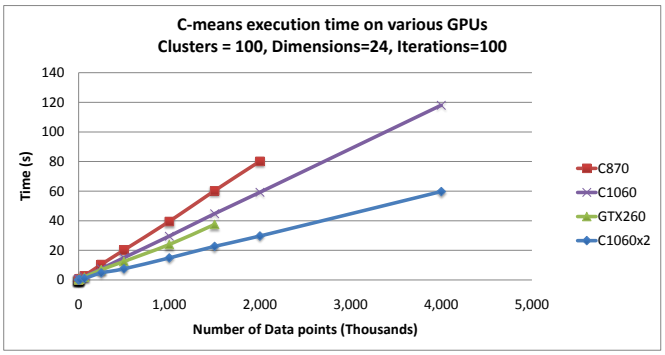

Fig. 3. C-means Execution Time

GTX 260) and the fourth using two Tesla C1060s. For $N > 10,000$ the GPU resources become fully utilized and speedup begins to level off.

The GTX 260 has the best performance of the three single-GPU configurations due to its superior global memory bandwidth. The algorithm performs relatively simple calculations (Euclidean distance calculations and weighted summations with exponentiation to update the cluster centroids) on a very large data set. The relatively low ratio of computation to memory access makes the algorithm inherently memory bound given the gap in processing performance and memory performance in modern computing systems, including GPUs. The Tesla C1060x2 configuration has an speedup of 1.99 for the kernel and 1.90 for the total execution time (including all overheads such as file I/O, host to device memory copying, and device initialization) compared to a single C1060.

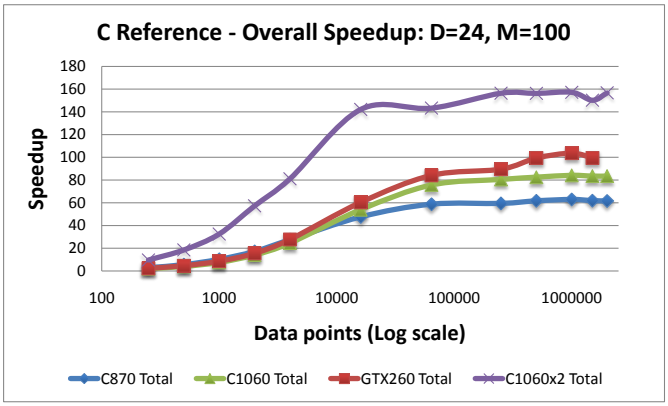

Fig. 4. C-means Speedup

Figure 5 shows the execution time for the Expectation Maximization with Gaussians implementation. As seen previously with C-means, a data set with 24 dimensions and 100 clusters was used along with four different GPU configurations. The execution time is linear for non-trivial data sets. As seen in Figure 6, speedup begins to level-off for data sizes of $N > 64,000$. The load balancing, global memory access patterns, and the use of shared memory is more complex in the EM algorithm, which leads to more variability (compared to C-means) in the speedup as the size of the data set changes.

TABLE I
C-MEANS KERNEL RESOURCE USAGE

Kernel	Blocks	Threads	Registers	SMEM	Occupancy
Distance	$M \times \lceil N/512 \rceil$	512	6	$D \times 4$	1.00
Membership	$\lceil N/512 \rceil$	512	7	0	1.00
Centers	$D \times \lceil M/4 \rceil$	256	12	4096	0.75
Sizes	M	512	6	2048	1.00

TABLE II
EM KERNEL RESOURCE USAGE

Kernel	Blocks	Threads	Register	SMEM	Occupancy
seed	1	256	15	$D \times 8$	1.0
estep1	$16 \times M$	512	16	$D^2 + D \times 4$	1.0
estep2	$2 \times \#SMs$	512	14	2048	1.0
mstep_n	M	256	8	1024	1.0
mstep_mean	$M \times D$	256	10	1024	1.0
mstep_covar	$\lceil M/6 \rceil \times D(D+1)/2$	256	29	6192	0.5
constants	M	32	14	$D^2 \times 4$	0.18

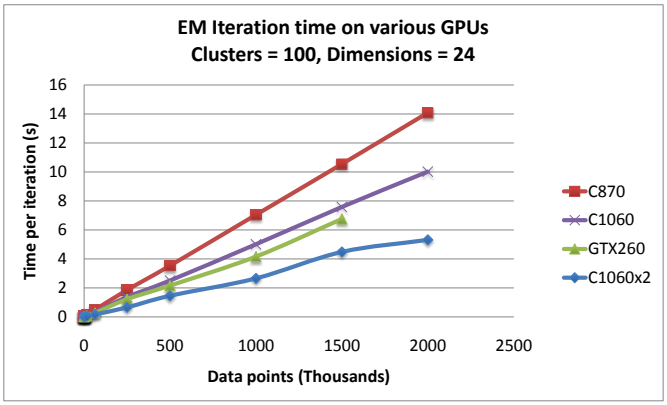

Fig. 5. Expectation Maximization Execution Time

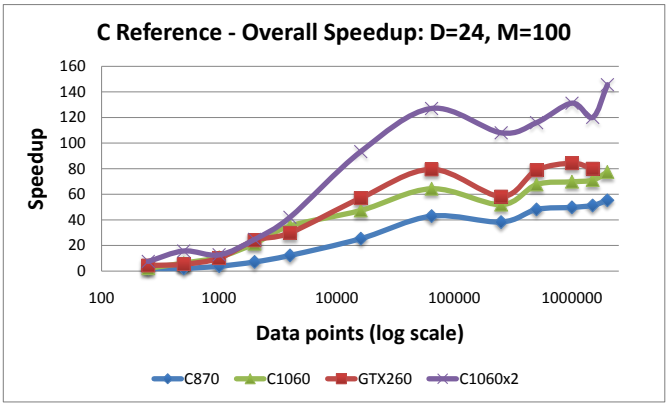

Fig. 6. Expectation Maximization Speedup

D. Comparison to Prior Work

The Background section of this paper surveyed the previous work on GPU-based acceleration of the C-means and EM with Gaussian algorithms. This section compares the performance of the current implementations in this paper to the previous work. Sometimes exact comparisons cannot be made due to differences in available hardware, but differences are described when relevant. Furthermore, the details provided in some of the prior papers do not give exact details about the experiments used for profiling their code. For example, some papers provide speedup figures compared to a proprietary CPU reference version. These numbers are not very meaningful without absolute time since an inefficient GPU algorithm can still have very good speedup if it is compared to a poor CPU version.

1) C-means: In 2009 Espenshade et al. published a paper on a single-GPU implementation of fuzzy C-means [13]. Speedups of 84x were observed, but the CPU version naively replicated many of the flaws in the GPU version. Table III shows a comparison of the current C-means implementation to the old implementation in [13] with identical problem sizes and the same testing hardware. The data size is 100K events with 16 dimensions. The new GPU version in this paper has an improvement over the previous GPU version ($t2/t3$) of 9x to 21x and would continue even higher with larger values of M. The biggest reason for the increase in performance was the reduction of the computational complexity of the implementation from $O(NDM^2)$ to $O(NDM)$. The new algorithm is not an approximation, it simply does not perform redundant distance and membership calculations like the older version did. The same improvement was made to the CPU reference version, which explains why the speedup values (t_1/t_2) differ significantly from the values in Table II of the old paper [13]. The new implementation also coalesces more global memory access and has higher kernel occupancy. Optimizations were also made to the center updates that offer considerably better scalability across a wide range of input parameters (the old implementation could not support more than about 32 dimensions, and gradually got slower with $D > 16$ due to resource constraints).

The next C-means paper is by Shalom et al. [12]. This paper is difficult to compare directly against since there are no iteration times nor are any specific details about the data given. It provides execution times, but without the number of iterations or the details of the synthetic data it is difficult to make an exact comparison. Shalom et al. used a NVIDIA 8800

TABLE III
COMPARISON OF C-MEANS EXECUTION TIME (MS) TO [13]

M	CPU CPU (t_1)	Espenshade et al. (t_2)	GPU (t_3)	Speedup (t_1/t_2)	Speedup (t_1/t_3)
4	92.4	11.88	1.27	7.8	73.1
8	183.15	15.35	1.74	11.9	107.2
12	267.62	21.61	2.05	12.4	130.6
16	339.89	26.36	2.56	9.3	133.2
24	540.03	51.40	4.07	10.5	132.8
32	728.08	60.69	5.67	12.0	125.3
48	1071.73	116.09	7.69	9.2	139.7
60	1353.69	204.36	9.72	6.6	138.7

GTX, which is similar in performance to the Tesla C870. Both are based on the same architecture and have 128 processing cores, but the 8800 GTX has 12% higher memory bandwidth.

In an attempt to compare to Shalom et al. a data set was generated with N=1048576, D=4, and four Gaussian clusters. The clusters have significant overlap in the first dimension (making the data set non-trivial) and become increasingly distinct in the remaining dimensions. A convergence criterion of $\epsilon = .00001$ was selected — the same as the Shalom et al. paper. Each GPU kernel iteration took 11.6 ms to complete. The worst case of 25 clusterings (with different randomly initialized centers) with the same data set took 21 iterations to converge. Total computational time (kernel and memory copying) was 0.27 seconds. Shalom et al. stated 0.91 seconds for the same data size and clustering parameters, but the number of iterations can vary depending on the data set.

Anderson et al. published two nearly identical papers on C-means — the first showing results for non-Euclidean distance metrics and the second focuses just on speedup with a standard Euclidean distance metric [10] [11]. The paper indicates that the testing was performed with 100 iterations on synthetic data with random cluster centers. Table IV compares the C-means performance to the numbers in the Anderson et al. paper with a Euclidean distance metric. Anderson et al. performance numbers were taken with a 8800 GTX whereas our numbers were using a Tesla C870 - a very similar albeit slightly slower GPU. The results show improvements ranging from of 1.04x to 4.64x. The improvement is particularly large for the two clustering profiles with 64 clusters and smallest for the two profiles with the largest number of dimensions. However, there is not enough timing data available to accurately determine exactly how the three parameters affect the improvement.

TABLE IV
COMPARISON OF C-MEANS EXECUTION TIME (MS) TO [11]

M	N	D	Anderson et al. (t_1)	C-means (t_2)	Speedup (t_1/t_2)
4	4096	4	39	12	3.25
4	4096	128	53	51	1.04
64	4096	4	172	53	3.24
64	8192	4	362	78	4.64
16	40960	32	471	331	1.42
4	409600	8	780	529	1.47

2) EM with Gaussians: Kumar et al. published a paper on Expectation Maximization with Gaussians in June 2009 [14]. Their implementation supports only diagonal covariance, so the version in this paper has been compiled for diagonal-only mode for the sake of this comparison. Kumar et al. provide per iteration performance results with a Quadro FX 5800. The FX 5800 is identical to the Tesla C1060 except that it is sold for workstations rather than servers and has display ports.

Table V shows the time per iteration from the Kumar et al. paper compared to the diagonal-only version of EM in this paper. The improvement is significant across all data sizes provided in the prior work and ranges from 3.72x to 10.1x. The covariance calculation is likely the biggest bottleneck for the Kumar et al. implementation (and it was listed in the paper's future work section). It solves for covariance by multiplying a very large variance matrix with the membership value matrix using the CUBLAS SGEMM. This technique computes the entire variance matrix and then extracts only the diagonal portion, which wastes a significant number of FLOPS.

TABLE V
COMPARISON OF EM KERNEL TIME (MS) TO [14]

N	M	D	Kumar et al. (t_1)	paper (t_2)	Speedup (t_1/t_2)
46.8K	8	8	16.2	1.6	10.1
76.8K	16	16	42.4	7.5	5.65
122.8K	24	24	102.1	24.6	4.15
153.6K	32	32	215.0	51.1	4.21
230.4K	32	32	264.9	71.1	3.72

Andrew Harp wrote an implementation of EM with Gaussians on CUDA with a MATLAB MEX wrapper as a project at the University of Texas in Austin [20]. The project website claims speedup up to 170x on a GTX 285 versus a Intel C2D E8400 operating at 3 GHz, but specific timing results are not given. Fortunately, the source code for the CPU reference version and GPU version is available online. The program requires MATLAB, which was not available on either of the CUDA testing machines used in this paper. However, a speedup comparison can be made by obtaining timing with the Harp CPU reference version and then calculating speedup with timing results from the paper implementations. Both CPUs are from the same generation (same Intel micro-architecture and manufacturing process), but differ in operating frequency by 10%.

The Harp experiment had 10 clusters, 8 dimensions, and varied the number of events from 1,000 to 1,000,000. Table VI compares the CPU time for the Harp CPU reference version to the version of EM in this paper with a GTX260 graphics card. The times in the table are for 100 iterations of EM and include GPU memory copying for the GTX260 column. The GTX260 has less cores (192 versus 240) and lower memory bandwidth (110 GB/s versus 159 GB/s) compared to a GTX 285 GPU. The implementation of EM in this paper provides 446x speedup compared to the Harp CPU reference. Harp reports a speedup of only 170x with a faster GPU. Therefore the improvement is at least 2.6x.

E. Overhead

CUDA applications use the GPU as a co-processor to accelerate the computations. This model of execution has inherent overheads that are not present in a traditional CPU-only solution. One of the primary bottlenecks for GPGPU

TABLE VI
COMPARISON OF EM TIME (S) TO [20]

N	M	D	Harp CPU Reference (t_1)	GTX260 (t_2)	Speedup (t_2/t_1)
1K	10	8	2.5	0.28	8.8
10K	10	8	14.5	0.29	49.5
100K	10	8	267	0.83	321.2
1000K	10	8	2772	6.20	446.7

applications is the transfer of data from the *host* memory to *device* memory, and then back to the *host* again after computing on the GPU. This section profiles the amount of time spent on different portions of the C-means and Gaussian mixture model implementations. The computation is broken down into 5 categories:

1) I/O — reading the data file from disk and generating/saving the result files
2) CPU Init — processing performed by the main CPU thread before splitting into one thread per GPU
3) CPU — processing performed by each *host* thread
4) GPU memcpy — *host-to-device* or *device-to-host* data transfer (memory copying)
5) GPU kernel — computational kernels on the emphdevice

Figure 7 shows the breakdown of the execution time for the C-means implementation and Figure 8 has a breakdown for EM with Gaussians. The C-means algorithm has a lower computational complexity than EM with Gaussians. Therefore the overhead such as reading the file from disk, initializing data structures on the host, and transferring data to and from the GPU, which is essentially the same for both algorithms, is a larger portion of the overall execution time for C-means. Note that the vertical axis is log-scale so that the overhead is more visible. For C-means the overheads together are approximately 10% of the execution time, compared to 1% for the Gaussian mixture model.

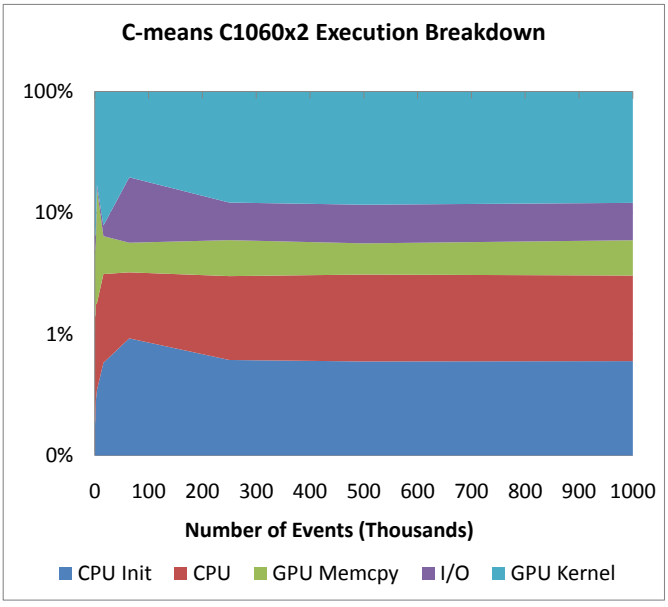

Fig. 7. Breakdown of Execution Time for C-means

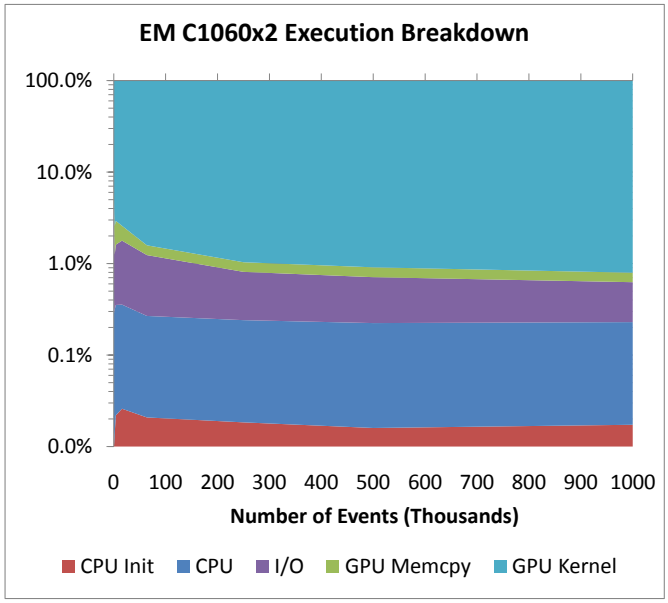

Fig. 8. Breakdown of Execution Time for Gaussian MIxture Model

The GPU kernel time is based on 100 iterations with two Tesla C1060 GPUs. Increasing the number of iterations affects the GPU kernel time and makes the overhead less significant. Therefore in reality simple data sets with distinct clusters that converge very quickly will not have as high of speedup as more complex data sets that require a large number of iterations, but 100 iterations is still a conservative estimate for large high dimensional data sets with a lot of clusters. Even with the simple synthetic data, there was often still significant cluster center movement after 100 iterations.

The data clustering algorithms in this paper are very suitable for a GPGPU applications because the data set only needs to be copied once to each GPU. Copying memory to and from the GPU has a very minmal impact on the total execution time in these implementations. I/O and CPU computation form the majority of the overhead in the implementation, which are present in the CPU-only implementation as well. Many iterations of computation occur with a very trivial amount of data transfer required for synchronizing the cluster parameters between multiple GPUs. The implementation also scales well to multiple GPUs because the data set gets dividedly evenly among all GPUs in the system and therefore the memory that must be copied to each device is inversely proportional to the number of GPUs.

F. Multi-Node

A primary objective of these implementations was to be scalable beyond a single workstation with a CUDA card to many nodes in a high performance computing environment, each with one or more CUDA capable accelerators. The results examine horizontal scalability — adding more independent computing nodes, as opposed to vertical scalability which improves the node(s).

The first set of results uses *fixed-problem size* (also often called *strong scaling*) analysis. This form of analysis keeps

the total amount of work constant (in this case the number of data points) and looks at the speedup compared to a single-node version of the same problem size. Analysis with a fixed-problem size is constrained by Amdahl's law [21] — any portion of the algorithm that is not parallelized will limit the maximum possible speedup of the algorithm even with infinite parallel resources. Figure 9 and Table VII shows the speedup of the C-means implementation from 1 to 64 nodes (2 to 128 Tesla C1060 GPUs). Although clustering large data sets can require billions of calculations, the actual computational complexity is linear with respect to any given input parameter. The overhead associated with file I/O, distributing the dataset to the different nodes, synchronizing results between each iteration, and collecting the final results becomes a bottleneck and reduces parallel efficiency with a fixed problem size. The implementation has 78% parallel efficiency with 16 nodes (32 GPUs) and only 42% efficiency with 64 nodes (128 GPUs).

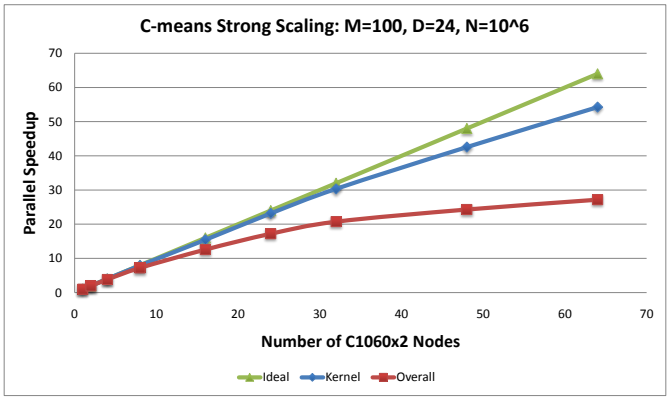

Fig. 9. Fixed Problem Size Speedup: C-means

TABLE VII
C-MEANS SPEEDUP SUMMARY

Nodes	M=100, D=24, N=1,000,000		
	GPU Kernel	GPU	CPU
1	1.00	1.00	176.39
2	1.98	1.95	343.99
4	4.00	3.84	676.83
8	7.92	7.22	1273.63
16	15.43	12.58	2219.09
24	23.16	17.21	3035.92
32	30.34	20.75	3660.04
48	42.56	24.31	4287.52
64	54.25	27.18	4794.94

Figure 10 and Table VIII show the fixed-problem size speedup results for the Expectation Maximization algorithm. EM has a higher computational complexity than C-means, $O(NMD^2)$, but the overhead for file I/O, dataset distribution, and result collection are nearly identical. Therefore a larger percentage of the total execution time can be parallelized and the parallel efficiency is higher. EM has 92% efficiency with 16 nodes and 72% efficiency with 64 nodes. With 128 GPUs the implementation has 6286x speedup compared to a single CPU (nearly 4 orders of magnitude) and can cluster a dataset with 1 million data points, 24 dimensions, and 100 Gaussian clusters in less than 10 seconds including all overheads except job queueing and setting up the MPI environment on the reserved computing nodes (which are an administrative overhead and not directly relevant to the algorithm performance).

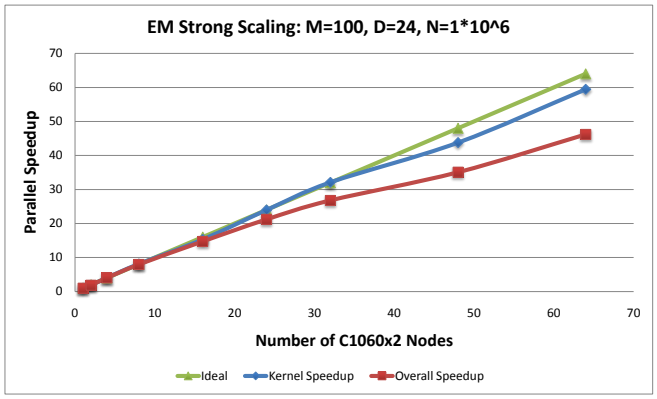

Fig. 10. Fixed Problem Size Speedup: Expectation Maximization

TABLE VIII
EXPECTATION MAXIMIZATION SPEEDUP SUMMARY

Nodes	M=100, D=24, N=1,000,000		
	GPU Kernel	GPU	CPU
1	1.00	1.00	136.04
2	1.87	1.87	253.75
4	4.07	4.03	548.29
8	8.12	7.96	1082.96
16	15.32	14.73	2004.30
24	23.98	21.22	2887.40
32	32.11	26.80	3646.55
48	43.79	35.08	4772.56
64	59.46	46.21	6286.48

TABLE IX
SCALED SPEEDUP SUMMARY: M=100, D=24, N=50K PER NODE

Nodes	C-means		EM	
	Kernel	Overall	Kernel	Overall
2	1.97	1.93	2.06	2.03
4	3.90	3.67	3.86	3.75
8	7.89	7.07	7.68	7.19
16	16.04	13.11	15.55	14.40
32	30.73	22.03	30.88	27.50
64	61.92	37.03	60.29	51.60

V. CONCLUSION

Data clustering algorithms have computational complexities that increase combinatorially as the dimensions of the input data and the number of clusters grow. Two very popular unsupervised data clustering algorithms are C-means (a fuzzy extension of K-means) and Gaussian mixture models optimized via expectation maximization. Both of these algorithms exhibit significant data parallelism and are good applications for high-performance parallel computing. The algorithms have previously been parallelized using a variety of parallel processing architectures including multi-core processors, commodity clusters, and high-performance supercomputing grids.

Modern GPU architectures are many-core systems with both high memory bandwidth and many more arithmetic resources

than modern CPUs. Although traditionally designed for graphics applications, GPUs are becoming increasingly capable of general purpose computing. Frameworks like NVIDIA's C-based CUDA have significantly improved the efficiency of accelerating general purpose applications on GPU hardware. Algorithms no longer have to be to cast into a graphics application using technologies like OpenGL and Cg.

This paper investigated the use of GPUs for accelerating these two popular unsupervised data clustering algorithms with NVIDIA's CUDA framework and Tesla GPU architecture. A technology used by biologists and immunologists for studying the characteristics of cells called flow cytometry creates large data sets on the order of 10^6 multivariate vectors with tens of dimensions which require clustering. The large data processing requirements of flow cytometry were a motivating factor for the research into the cost-effective computing power of GPUs for data clustering. The research shows that GPUs are in fact very suitable for handling data clustering tasks on flow cytometry data sets. The flow cytometry results themselves with these two algorithms may not be ideal, but the research in this paper presents a suitable framework for acceleration of different clustering algorithms on the GPU with both single workstations and HPC environments. Speedup on flow cytometry sized data sets with a NVIDIA GTX 260 (a mid-range consumer graphics card) are 106x for C-means and 73x for EM with Gaussians compared to optimized C versions running on a single core of a modern Intel CPU.

Due to the popularity of these clustering algorithms, and the surge of research efforts to take advantage of the massive computational power of GPUs for general purpose applications, there have been previous efforts to accelerate the C-means and EM algorithms. GPGPU research has a history of comparing GPU-enhanced versions of algorithms to naive inefficient CPU reference code and boasting huge speedup figures. Comparisons must be made using absolute execution time rather than speedup to accurately compare multiple GPU implementations. This paper makes significant improvements ranging from 1.5x to 10x compared to the previous work using single-GPU implementations with similar hardware.

In addition to improvements in the single GPU implementations, both clustering algorithms have been expanded to use a hybrid of OpenMP and MPI. This allows the program to leverage multiple GPUs on a single machine as well as multiple nodes in a cluster environment. The National Center for Supercomputing Applications (NCSA) at the University of Illinois Urbana-Champaign has a Tesla-enhanced supercomputer called Lincoln. Using Lincoln as a testing environment, the hybrid MPI+OpenMP+CUDA clustering algorithms were tested for scalability and speedup.

C-means, with its lower computational complexity, does not scale as well as EM with Gaussians since overhead begins to dominate the execution time earlier. Even still, the C-means implementation achieves a parallel speedup efficiency of 79% with 32 GPUs with a speedup of 2219x compared to a single CPU with a data size on the order of a single flow cytometry file. Increasing the data size improves the efficiency. Using data sizes four times larger (4×10^6) the algorithms scales to 32 GPUs with 85% efficiency and a speedup over the CPU of 2368x. With 128 GPUs the efficiency falls to 53% with a CPU speedup of 5915x. Expectation Maximization with Gaussians has an efficiency of 72% with 128 GPUs using the smaller data set (1 million events with 24 dimensions) with a CPU speedup of 6286x.

VI. Future Work

This paper focused on using GPUs as co-processors to accelerate a couple of popular clustering algorithms. There are many other clustering algorithms that could be accelerated with GPUs and applied to the flow cytometry problem using the same hybrid of CUDA, OpenMP, and MPI technologies. Skewed T-mixtures have shown promise in previous research [22] for providing good clustering of flow cytometry's complex cluster shapes, but it has a computational complexity that is even higher than a Gaussian mixture model and is impractical on a single CPU.

The two algorithms in this paper still have some weaknesses that could be improved. Many areas of both algorithms exploit the data parallelism of individual clusters, but this comes at the cost of repeatedly accessing the input data. Similarly, exploiting the data parallelism of multiple dimensions comes at the cost of accessing membership values multiple times. Some of the kernels can be replaced by SGEMM (matrix multiplication) followed by a simple normalization, which is a more efficient way of performing the same calculation. Unfortunately an SGEMM that operates more efficiently on non-square matrices is required (CUBLAS SGEMM is designed for square matrices). A new implementation of SGEMM or 3rd party alternatives to NVIDIA's CUBLAS such as MAGMA or CULAtools could be investigated.

One current limitation of the MPI implementation in this paper is the master-slave structure. The root node needs to be able to hold the entire input data and the entire result matrix. Therefore the scalability of the algorithm to very large data sets is limited by the memory of the root node. A more decentralized approach where each node retrieves the input data independently and writes its membership results to an output would remove this limitation. MPI-IO is one possible solution. Other areas of exploration include, but are not limited to:

- Use the CPU cores for the kernels in addition to just the GPUs. Data points could be allocated to the CPU cores based on the relative performance between each CPU core and the GPU.
- Dynamic load balancing for more heterogeneous environments, such as GPUs with different performance on a single node or cluster nodes with mixed numbers/types of GPUs
- Re-evaluate parallel algorithm design for new GPU architectures such as NVIDIA Fermi which has a caching hierarchy for global memory. May be able to take kernels previously limited by shared memory and make them more coarse-grained (and therefore iterating over the input data fewer times), such as the C-means Update-Centers, EM E-step, and EM constants kernels.

ACKNOWLEDGMENT

This research was supported in part by the National Science Foundation through TeraGrid resources provided by the National Center for Supercomputing Applications at the University of Illinois Urbana-Champaign under grant number TG-MIP050001. Work conducted by Gregor von Laszewski is supported (in part) by NSF CMMI 0540076 and NSF SDCI NMI 0721656.

REFERENCES

[1] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, 1999.

[2] G. Gan, C. Ma, and J. Wu, *Data Clustering Theory, Algorithms, and Applications*, M. T. Wells, Ed. Society for Industrial and Applied Mathematics, 2007.

[3] NVIDIA. Cuda zone. [Online]. Available: www.nvidia.com/cuda

[4] 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.

[5] "Nvidia's next generation cuda compute architecture: Fermi," NVIDIA, 2009. [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html

[6] *NVIDIA CUDA Programming Guide*, NVIDIA, 2009. [Online]. Available: http://developer.nvidia.com/object/cuda_2_3_downloads.html

[7] *NVIDIA CUDA C Programming Best Practices Guide*, 2nd ed., NVIDIA, 2009. [Online]. Available: http://developer.nvidia.com/object/cuda_2_3_downloads.html

[8] H. Takizawa and H. Kobayashi, "Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing," *J. Supercomput.*, vol. 36, no. 3, pp. 219–234, 2006.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370 – 1380, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/B6WKJ-4SVV8GS-2/2/f7a1dccceb63cbbfd25774c6628d8412

[10] D. Anderson, R. H. Luke, and J. M. Keller, "Incorporation of non-euclidean distance metrics into fuzzy clustering on graphics processing units," *Analysis and Design of Intelligent Systems using Soft Computing Techniques*, vol. 41, pp. 128–139, 2007.

[11] D. Anderson, R. Luke, and J. Keller, "Speedup of fuzzy clustering through stream processing on graphics processing units," *Fuzzy Systems, IEEE Transactions on*, vol. 16, no. 4, pp. 1101–1106, Aug. 2008.

[12] S. A. A. Shalom, M. Dash, and M. Tue, "Graphics hardware based efficient and scalable fuzzy c-means clustering," in *AusDM*, ser. CRPIT, J. F. Roddick, J. Li, P. Christen, and P. J. Kennedy, Eds., vol. 87. Australian Computer Society, 2008, pp. 179–186.

[13] J. Espenshade, A. Pangborn, G. von Laszewski, D. Roberts, and J. Cavenaugh, "Accelerating partitional algorithms for flow cytometry on gpus," in *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, Aug. 2009, pp. 226–233.

[14] N. Kumar, S. Satoor, and I. Buck, "Fast parallel expectation maximization for gaussian mixture models on gpus using cuda," in *11th IEEE International Conference on High Performance Computing and Communications, 2009. HPCC'09*, 2009, pp. 103–109.

[15] Y. Tarabalka, T. Haavardsholm, I. Kåsen, and T. Skauli, "Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and gpu processing," *Journal of Real-Time Image Processing*, vol. 4, no. 3, pp. 287–300, 2009.

[16] M. A. Suchard, Q. Wang, C. Chan, J. Frelinger, A. Cron, and M. West, "Understanding gpu programming for statistical computation: Studies in massively parallel massive mixtures," *Journal of Computational and Graphical Statistics*, vol. 19, no. 2, pp. 419–438, 2010.

[17] C. A. Bouman, "Cluster: An unsupervised algorithm for modeling gaussian mixtures," April 1997, available from http://www.ece.purdue.edu/~bouman.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2007.

[19] N. C. for Supercomputing Applications, "Ncsa intel 64 tesla linux cluster lincoln technical summary," 2010. [Online]. Available: http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/TechSummary/

[20] A. Harp, "Em of gmms with gpu acceleration," May 2009. [Online]. Available: http://andrewharp.com/gmmcuda

[21] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA: ACM, 1967, pp. 483–485.

[22] S. Pyne, X. Hu, K. Wang, E. Rossin, T. Lin, L. Maier, C. Baecher-Allan, G. McLachlan, P. Tamayo, D. Hafler *et al.*, "Automated high-dimensional flow cytometric data analysis," *Proceedings of the National Academy of Sciences*, vol. 106, no. 21, pp. 8519–8524, 2009.

APPENDIX

Supplemental materials, including all source code and the data used to generate the Figures in the results section, are available online at http://apangborn.com/cuda_clustering.