Scalable Data Clustering using GPUs

by

Andrew D. Pangborn

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Associate Professor Dr. Muhammad Shaaban
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2010

Approved by:

Dr. Muhammad Shaaban, Associate Professor
Thesis Advisor, Department of Computer Engineering

Dr. Gregor von Laszewski, Assistant Director of Cloud Computing
Committee Member, Pervasive Technology Institute, Indiana University

Dr. Roy Melton, Lecturer
Committee Member, Department of Computer Engineering

Dr. James Cavenaugh, Postdoctoral Fellow
Committee Member, Center for Vaccine Biology and Immunology, University of Rochester
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Scalable Data Clustering using Graphics Processing Units

I, Andrew D. Pangborn, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Andrew D. Pangborn

Date
Dedication

To my family Don, Merry, and Alicia, for their never-ending support, encouragement, and patience
I am grateful for Gregor von Laszewski who sparked my interest in parallel computing, inspired this thesis project, and provided continual research guidance and professional advice. Special thanks to Dr. James Cavenaugh and his colleagues in the Flowgating group at the University of Rochester Center of Vaccine Biology and Immunology for their expertise and data.

Finally, I would like to thank my advisors Dr. Muhammad Shaaban and Dr. Roy Melton whose instruction and guidance in the fields of computer architecture and scientific programming has been invaluable with this thesis and my own development as a professional.

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.
Abstract
Scalable Data Clustering using GPUs

Andrew D. Pangborn

Supervising Professor: Dr. Muhammad Shaaban

Flow cytometry is a mainstay technology used by biologists and immunologists for counting, sorting, and analyzing cells suspended in a fluid. Like many modern scientific applications, flow cytometry produces massive amounts of data, which must be clustered in order to be useful. Conventional analysis of flow cytometry data uses manual sequential bivariate gating. However, this technique is limited in the quantity, quality, and speed of analyses produced. Unsupervised multivariate clustering techniques have shown promise for producing sound statistical analyses of flow cytometry data in previous research.

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.
## Contents

**Dedication** ......................................................... iii

**Acknowledgments** ............................................... iv

**Abstract** .............................................................. v

1 **Introduction** .................................................. 1

2 **Background** .................................................... 3
   2.1 Data Clustering .................................................. 3
      2.1.1 Types of Data clustering ................................. 3
   2.2 Flow Cytometry ................................................ 5
      2.2.1 Data ...................................................... 6
      2.2.2 Bivariate Gating Analysis .............................. 7
      2.2.3 Multivariate Analysis ................................. 8
   2.3 Workflow ....................................................... 8
      2.3.1 Data Extraction ......................................... 9
      2.3.2 Data Preparation ....................................... 10
      2.3.3 Clustering ............................................... 11
   2.4 GPGPU ............................................................ 12
      2.4.1 CUDA .................................................... 14

3 **Supporting Work** ............................................. 21
   3.1 Algorithms ................................................... 21
      3.1.1 C Means ................................................ 21
      3.1.2 Gaussian Mixture Models .............................. 22
   3.2 Data Clustering on GPGPU .................................. 25

4 **Parallel Algorithm Implementations** ....................... 28
   4.1 Parallel Implementation Architecture ....................... 28
   4.2 C-means ....................................................... 31
      4.2.1 Data Organization ....................................... 34
      4.2.2 Distance Kernel ........................................ 34
4.2.3 Membership kernel ............................................. 36
4.2.4 Update Centers kernel ....................................... 39
4.2.5 Comparison to prior work ................................. 44

4.3 Expectation Maximization ............................... 45
4.3.1 Data organization .............................................. 46
4.3.2 Initialization .................................................... 48
4.3.3 E-step ............................................................. 48
4.3.4 M-step ............................................................. 49
4.3.5 Combine Gaussians ........................................... 54
4.3.6 Summary .......................................................... 54

5 Results ................................................................. 56
5.1 Testing Environment ........................................... 56
5.2 Resource Utilization ........................................... 57
5.3 Single Node Performance ................................. 60
5.3.1 Algorithm Complexity ......................................... 60
5.3.2 Speedup ............................................................. 69
5.4 Comparison to prior work ........................................ 80
5.4.1 C-means ............................................................ 80
5.4.2 EM with Gaussians ............................................. 82
5.5 Horizontal Scalability ............................................. 83
5.5.1 Fixed Problem Size Scalability ....................... 84
5.5.2 Time-constrained Scaling ................................. 90
5.6 Results - Iris Data .................................................. 92
5.7 Verification - Synthetic Data ............................. 96
5.8 Flow Cytometry Data ............................................. 102

6 Conclusion ................................................................. 109

7 Future Work ............................................................... 111

Bibliography ............................................................... 113
List of Tables

4.1 UpdateCenters vs. CUBLAS SGEMM ........................................ 44

5.1 C-means Kernel Resource Usage ........................................... 58
5.2 C-means Kernel FCS File Resource Usage ............................... 59
5.3 EM Kernel Resource Usage .................................................... 59
5.4 EM Kernel Shared Memory Usage ......................................... 59
5.5 EM Kernel Performance with Different Threads per Block .......... 60
5.6 Comparison of C-means Execution time (ms) to [1] .................... 80
5.7 Comparison of C-means Execution time (ms) to [2] .................... 81
5.8 Comparison of EM Kernel time (ms) to [3] ............................ 82
5.9 Comparison of EM time (s) to [4] ......................................... 83
5.10 C-means Speedup Summary .................................................. 88
5.11 Expectation Maximization Speedup Summary ........................... 90
5.12 Scaled Speedup Summary: M=100, D=24, N=50K per node ........ 94
5.13 Iris Data Cluster Sizes ....................................................... 94
5.14 Iris Data Cluster Means ...................................................... 96
5.15 Synthetic Data Test 1: C-means ............................................. 98
5.16 Synthetic Data Test 1: Gaussians ......................................... 99
5.17 Test #2 Dataset .............................................................. 100
5.18 Test #2 C-means Clustering Summary .................................... 101
5.19 Test #2 EM Gaussian Clustering Summary ............................. 101
5.20 Human/Mouse Flow Cytometry Data Variables ....................... 103
# List of Figures

2.1 Taxonomy of Clustering Algorithms [5] .............................................. 4  
2.3 Bivariate Scatter Plot [6] ............................................................. 8  
2.4 Flow Cytometry Workflow .............................................................. 9  
2.5 NVIDIA Tesla G80/G92 GPU Architecture [7] .................................. 13  
2.6 NVIDIA CUDA Architecture Overview [8] ....................................... 15  
2.7 CUDA Multiprocessor Architecture [9] ............................................ 16  
2.8 CUDA Thread Hierarchy [9] ............................................................. 16  
2.9 CUDA Memory Model [9] ................................................................. 18  

3.1 High-Level Clustering Procedure [10] ............................................... 25  

4.1 Process Hierarchy .............................................................. 29  
4.2 Communication and Reduction Hierarchy ........................................... 30  
4.3 Pseudo-Code for the Program Flow of C-means on the Host ................. 33  
4.4 Pseudo-code for Distance Kernel ..................................................... 35  
4.5 Effect of Grid Size on Distance and Membership Kernels .................. 36  
4.6 Pseudo-code for Membership Kernel ............................................... 37  
4.7 C-means Partition Camping ............................................................ 38  
4.8 Pseudo-code for Membership Kernel ............................................... 39  
4.9 Pseudo-code for UpdateCenters kernel ........................................... 41  
4.10 CUBLAS implementation of Cluster Centers ................................... 43  
4.11 CUBLAS SGEMM Performance on 2 data sets ................................ 43  
4.12 Gaussian mixture model data structure .......................................... 47  
4.13 E-step log-likelihood kernel pseudo-code ....................................... 49  
4.14 E-step membership kernel pseudo-code ......................................... 50  
4.15 M-step host pseudo-code .............................................................. 51  
4.16 M-step \( n \) pseudo-code ............................................................. 51  
4.17 M-step \( \mu \) pseudo-code ............................................................. 52  
4.18 M-step covariance pseudo-code ...................................................... 53  

5.1 C-means total time scalability: \( N \) ..................................................... 62
5.2 C-means GPU kernel scalability: N .................................................. 62
5.3 Expectation Maximization Scalability: N ........................................... 63
5.4 Breakdown of Execution Time for C-means and EM: N ...................... 64
5.5 Naive C-means Scalability: M ....................................................... 65
5.6 Optimized C-means Scalability: M ................................................... 65
5.7 Expectation Maximization Scalability: M ........................................... 66
5.8 Breakdown of execution time for C-means and EM: M ....................... 66
5.9 C-means Scalability: D ............................................................... 68
5.10 Expectation Maximization Scalability: D ........................................... 68
5.11 Breakdown of execution time for C-means and EM: D ....................... 69
5.12 C-means Kernel Speedup vs. events: C reference version .................. 70
5.13 C-means Speedup vs. events: C reference version ............................ 70
5.14 C-means Speedup vs. events: MATLAB fcm ................................... 72
5.15 C-means Speedup vs. dimension: C reference ................................. 72
5.16 C-means Speedup vs. dimension: MATLAB .................................... 73
5.17 C-means Speedup vs. clusters: C reference ..................................... 74
5.18 C-means Speedup vs. clusters: MATLAB ........................................ 74
5.19 EM Kernel Speedup vs. events: C reference version ....................... 75
5.20 EM Speedup vs. events: C reference version .................................... 75
5.21 EM Speedup vs. events: MATLAB Gmdistribution.fit EM ................. 77
5.22 EM Speedup vs. dimension: C reference ........................................ 77
5.23 EM Speedup vs. dimension: MATLAB ........................................... 78
5.24 EM Speedup vs. clusters: C reference ............................................ 79
5.25 EM Speedup vs. clusters: MATLAB ............................................... 79
5.26 Amdahl’s law with various degrees of parallelization ...................... 84
5.27 C-means Strong Scaling Speedup ................................................... 85
5.28 C-means Strong Scaling Time Breakdown ...................................... 86
5.29 C-means Strong Scaling Speedup ................................................... 87
5.30 EM Strong Scaling Speedup .......................................................... 89
5.31 EM Strong Scaling Time Breakdown .............................................. 89
5.32 C-means execution time with constant problem size per node .......... 91
5.33 EM execution time with constant problem size per node .................. 92
5.34 C-means Scaled Speedup ............................................................ 93
5.35 EM Scaled Speedup ................................................................. 93
5.36 Iris Data Set [11] .......................................................................... 95
5.37 Test 2 Gaussian Mixture Data ....................................................... 100
5.38 Test 2 MDL Scores ................................................................. 102
5.39 Density Plot of Forward Scatter vs. Side Scatter with a Physical Mixture of 50% Human Blood Cells and 50% Mouse Blood Cells ............................. 104
5.40 Density Plot of an electronic mixture (bottom) of Human blood cells (top-left) and Mouse blood cells (top-right) with hCD45 and mCD4 variables ..................... 105
5.41 Effect of the Number of Clusters on Classification of Human and Mouse Cells . . 106
5.42 Plot of 16 clusters with a 50% mixture of human and mouse cells: hCD45 vs. mCD4 107
Chapter 1

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 [5] [12]. 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 [12].

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.
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 thesis 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 [13]. The existing work on data clustering algorithms using GPGPU has been limited in the algorithms implemented, the scalability of such algorithms (such as using multiple GPUs), and lack of optimization specifically for the flow cytometry problem. 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 thesis is organized as follows. Chapter 2 provides an overview of data clustering, flow cytometry, modern GPU architectures, and the CUDA framework. After motivating the use of GPUs for data clustering, Chapter 3 discusses the specific clustering algorithms implemented in the thesis followed by previous GPGPU efforts with clustering algorithms. Chapter 4 provides implementation details for the parallel clustering algorithms and discusses improvements over previous work. Chapter 5 analyzes results for the parallel clustering algorithms using a variety of different performance metrics. Finally, Chapters 6 and 7 conclude the thesis and provide suggestions for future work.
Chapter 2

Background

This chapter begins with an overview of data clustering and discusses the specific classes of clustering algorithms used in this thesis. The basics of flow cytometry and the characteristics of the data are covered to motivate the use of GPUs for high performance clustering of such data sets. Finally the chapter provides an overview of the NVIDIA Tesla GPU architecture and CUDA framework such that the reader can understand the implementation descriptions in Chapter 4.

2.1 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. No single data clustering algorithm is sufficient for all applications.

2.1.1 Types of Data clustering

An exhaustive discussion of data clustering techniques is beyond the scope of this thesis. However, this section provides an overview of the different types of data clustering algorithms that are accelerated by GPUs in this thesis. For a more thorough review of data clustering techniques please consult [5] and [12].

Data analysis can be dichotomized into two broad efforts — exploratory or confirmatory [5]. The former case attempts to gain insight into the contents of the data and form hypotheses, whereas the latter attempts to match the data to known patterns or models. Data clustering tends to fall into
the former case, whereas the latter cases are categorized as classification techniques. Flow cytometry has a wide variety of applications that fall into both categories. Regardless of the technique involved, data clustering algorithms face the same questions. How many meaningful clusters exist in the data? How well does the data fit the cluster configuration or model? Are all of the natural clusters being exposed? What does each cluster represent? While philosophically simple, in practice these are difficult problems, and as such these sorts of questions have plagued researchers for decades and are the reason why data clustering remains such a challenging and computationally intensive problem.

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.

Among the unsupervised (exploratory) clustering techniques are many subcategories. Figure 2.1 displays a taxonomy of clustering techniques from Jain et al. [5]. In this thesis three different types of data clustering are investigated.

- Square error: Fuzzy C-means
- Mixture Resolving: EM with Gaussians
- Hierarchical: Top-down Agglomerative Gaussians

![Figure 2.1: Taxonomy of Clustering Algorithms [5]](image-url)
The first is a partitioning method that minimizes square error [5] — although some literature classifies it as a center-based method [12]. In center-based techniques, a configuration with $k$ clusters has $k d$-dimensional cluster centers. The most common center-based clustering method is the $k$-means algorithm. Events in the data set are grouped into the cluster whose center is closest. Centers are then recomputed based upon their members. The center-based methods fall into the category of squared error clustering because it is minimizing the distance (error) between the data vectors and the nearest cluster. Fuzzy K-means [14] (typically called C-means in the data clustering literature) is an extension of K-means that computes membership values for each cluster as opposed to a hard classification of each vector to a single cluster. C-means is implemented in this thesis and will be discussed in more detail in the following chapter.

The next method is a mixture resolving (model-based) partitioning method. It assumes that the data are composed of a collection (mixture) of different distributions. The model-based algorithm investigated in this thesis is expectation maximization (EM) with Gaussians. It is a two-step iterative clustering algorithm, where the first step of each iteration maps the probability of each individual data point to the current models, and the second stage updates the statistical parameters of the models based upon the new probabilities.

Agglomerative hierarchical clustering works by computing pairwise distances between all objects and then iteratively combining the most similar objects. The similarity measure varies significantly from one implementation to another. This thesis does not use hierarchical clustering to cluster all of the data. Instead it combines hierarchical clustering with the Gaussian Mixture Model approach. In this case, the Gaussian models are combined rather than the individual data elements — in other words, the clusters are being clustered. The similarity measure is based upon the covariances and means of the Gaussian distributions.

### 2.2 Flow Cytometry

Flow cytometry is a process for studying the physical and chemical characteristics of a group of particles (usually cells) suspended in a fluid. The system contains three major components — fluidics, optics, and electronics. The fluidics system is responsible for creating a narrow stream of particles that pass through the optics by a technique known as hydrodynamic focusing (see Figure 2.2) [15]. Optics consist of one or more lasers of different wavelengths. Lenses, mirrors, and filters direct the scattered light to different photo-sensitive electronic detectors as seen in Figure 2.2. The particles (typically cells) pass through a focused laser beam at a rate of thousands per second. Light hits the cell and is refracted (scattered) in all directions. Forward scatter information can describe physical characteristics of the particle, such as cell size. Other light gets scattered to the side, and these rays are channelled through a series of filters for different wavelengths and are reflected onto other light detectors. A typical way of studying different cell characteristics is to
use fluorescently labeled antibodies. The cell samples are doped with various fluorescent reagents, which bind to certain types of cells, to act as indicators. When light hits the fluorescent molecules, they become excited and emit light at a particular wavelength [6]. Different colors of light will be emitted depending upon which fluorescent markers are attached to the cell, and the corresponding light emission is picked up by a series of detectors. [15]

![Figure 2.2: Hydrodynamic Focusing of Particles in a Flow Cytometer [6]](image)

### 2.2.1 Data

The information from all of the different detectors forms a data vector for each event — a particle or cell passing through the laser. Current optical flow cytometers can have as many as 20 detectors. Emerging state of the art instruments such as CyTOF [16] currently have 35 dimensions and are expected to grow to 50 to 60 dimensions in the near future. Flow cytometers produce large data sets on the order of $10^5$–$10^6$ events with tens of dimensions. The computational complexity of both algorithms grows combinatorially with respect to the dimensions of the input and the number of clusters. For example, 1 million events with 24 dimensions and 100 clusters would require 2.4 billion distance calculations per iteration of C-means. These large data sets motivate the use of high performance computational architectures such as GPU co-processors.

In addition to the raw data there are headers to indicate to which color/sensor the dimensions correspond, and compensation information for the different channels. Annotation information for the cell sample can also be present. The industry uses the flow cytometry standard (FCS) file format which contains header, compensation data, annotation information, and raw data [17].
2.2.2 Bivariate Gating Analysis

Analyzing multivariate data with more than a few dimensions is challenging because it is difficult to visualize or summarize the data. Bivariate gating is the traditional and most mature, albeit limited, approach used in the flow cytometry field for analyzing data. Histograms plot a single dimension of the data versus the frequency of a range in the cell population. It is fairly reasonable for an operator to observe 10-20 histogram plots, but this visualization technique does not allow the analyst to observe all of the overlapping, but naturally distinct, cell populations in the data. Instead, two dimensions at a time are analyzed in a scatter plot to expose more distinct cell populations. Figure 2.3 shows a simple scatter plot of the forward and side refraction data for a flow cytometry sample and their corresponding histograms. The different colors in the plot denote the natural clusters, (in this case, different types of cells such as lymphocytes, monocytes, and neutrophils) [6]. Naturally the cell type for each event would not be known a priori with real data, and therefore the analyst would group the blobs of cells in the scatter plot into distinct clusters. Expert knowledge must then be used by the immunologist studying the data to determine to what each cluster corresponds based upon the dimension and location of the clusters in the plot. For non-trivial analyses it is necessary to cluster the data iteratively rather than to use single scatter plots. Bins are drawn around cell populations, and then cells within a particular bin are further analyzed using different dimensions. This technique is known as sequential bivariate gating.

There are many software packages available for flow cytometry, such as FlowJo [18] and WebFlow [19]. However, most of this software focuses on bivariate analysis techniques. Services provided include data management and organization, statistical analysis, gating, and visualization.

2.2.3 Multivariate Analysis

The previous section discussed how moving from one dimension to two dimensions for analysis exposed many more natural clusters in the data. However with data containing upwards of 24 dimensions, there is still a lot of potential information in the data that may not be exposed by bivariate techniques. As the number of colors increases, the parameter space for bivariate analysis grows rapidly. While a simple experiment with 6 colors has only 64 possible boolean parameters, a 20-color experiment has on the order of $10^6$.

Using an iterative sequential bivariate technique with multiple scatter plots and different combinations of dimensions allows the analyst to see a lot of different characteristics in the cell population. However, the number of different ways of analyzing the dimensions two at a time grows combinatorially, and consistent, reliable data analysis is difficult for manual operators. Very slight changes in the binning in one iteration of the sequential bivariate analysis can also have a large impact on the final result.

Instead, this thesis investigates the use of multivariate data clustering techniques for analyzing
flow cytometry data. Despite the presence of automatic multivariate clustering in the flow cytometry literature for nearly 25 years [20] [21] [22], it has had little effect on the flow cytometry community in practice. However, some recent research efforts such as FLAME [23] and Lo et. al [24] have shown promise for the use of automated multivariate clustering for flow cytometry with biological case studies. Only recently have any multivariate flow cytometry data analysis tools began to emerge for widespread use, such as Immport [25], and FlowCore [26]. Many of the methods have high computational complexity and it is difficult for researchers to analyze flow data using these methods in a timely fashion on a CPU.

### 2.3 Workflow

So far Chapter 2 has discussed the characteristics of flow cytometry data and a brief overview of an individual analysis of the data. Although this thesis focuses on the computational bottleneck of unsupervised multivariate clustering, the workflow for analyzing flow data generally requires more than a single clustering of the raw data from the flow cytometer. This section describes the pre-processing steps required before the flow data set is clustered on the GPU, the clustering itself, and analysis of the results.
A suite of Python scripts were developed for the pre-processing stages of the workflow to prepare flow data for clustering by the CUDA program. Additional scripts generate plots from the clustering result files. These scripts utilize NumPy, which is the de-facto open-source library for working with matrices and numeric processing in Python, and Matplotlib for plotting.

![Flow Cytometry Workflow Diagram](image)

Figure 2.4: Flow Cytometry Workflow

### 2.3.1 Data Extraction

Data collected by flow cytometers are stored in the Flow Cytometry Standard (fcs) format. It contains a header, a text portion with meta-data (instrument details, compensation, patient/sample information, etc.), and the raw data. The complete details of the FCS 3.1 format are maintained by the International Society for Advancement of Cytometry [17].

The first script in the workflow extracts the relevant information from the binary FCS file. There are two key pieces that must be extracted from the FCS file for the clustering workflow — the SPILL variable information (found in the TEXT portion of the FCS file) and the raw data (found in the DATA portion of the fcs file). SPILL contains a list of the fluorescent variables as well as a matrix of spillover information. This information is used to try to negate the effects of fluorescent spillover (an inherent side-effect of the optics used in the machines) from one channel to another. The process of negating the spillover effect is called data compensation. If compensation is performed properly by the operator, it results in data that more accurately reflects the underlying biological responses and more uniform clusters.
2.3.2 Data Preparation

After the data and spillover information have been extracted, the data are filtered, compensated, and transformed. Filtering removes data points where the light detectors have been saturated (equal to the maximum value of the ADC on the instrument). Due to saturation these values cannot be accurately measured. They are removed before clustering to prevent results being skewed by values accumulated on an artificial axis (the upper-bound of the machine’s measuring capability). A high percentage of saturated values indicates that the machine voltages and gains are not properly configured.

The next step is to compensate the data using the spillover matrix. Often times the spillover is not contained within the FCS file itself, but provided separately — the script supports both scenarios. The raw data (dimensionality $N \times D$) found in the FCS file are the product of the compensated data (the data of interest) and the spillover matrix (dimensionality $D \times D$). Thus the compensated data can be calculated by multiplying the raw data by the inverse of the spillover matrix, (i.e. the compensation matrix).

\[
raw = compensated \ast spillover
\]
\[
compensated = raw \ast spillover^{-1}
\]

Transformation is the final step. Flow data contain two main classes of variables — scatter and fluorescent. Scatter information measures how much the light scatters (or spreads) when the laser hits the cell. This results in relatively linear data and does not require any transformation (nor is it included in the compensation). These values range from 0 to $2^{18} - 1$, (the machine has an 18-bit ADC), for the data files used in this thesis from the University of Rochester Center of Vaccine Biology and Immunology. The fluorescent variables are the key biological indicators and tend to exhibit log-like responses. The two unsupervised data clustering methods used in this thesis rely on the dimensions in question to be weighted relatively evenly in order to work properly. Therefore all fluorescent dimensions are transformed from a log-like scale to a linear scale, and then the data set is rescaled to match the range of the scatter variables. Manual gating software effectively analyzes data the same way since fluorescent variables are plotted on a log scale.

The Logicle (a form of the biexponential) transformation [27] is often referenced in flow cytometry research and used in popular flow cytometry tools such as FlowJo [18]. This transformation addresses the problem of log data with both negative and positive values and the often non-uniform shapes encountered in flow data. The transformation is linear near zero and becomes logarithmic for larger values, resulting in a relatively linear display of the originally log-like data across the entire range.

Filtering, compensation, and transformation are all performed by a single script. The Logicle
transformation implementation is based on a MATLAB application written by Jonathan Rebhahn at the University of Rochester. The data can be written to a comma-separated-value (CSV) file after all data pre-processing is complete. CSV is a very generic and portable format that allows the clustering applications to maintain a simple interface for any type of floating-point input data. It also allows the data to be interpreted by other programs such as MATLAB or plotting utilities. However, processing large CSV files is computationally intensive, which can be prohibitive for parallel speedup (assuming the I/O is not parallel). Therefore the scripts also output to a simple binary file format. Using binary files reduced I/O time for the CUDA programs by an order of magnitude.

The first four bytes of the binary file are a 32-bit integer encoded in little endian that stores the number of events in the file. The next four bytes are an integer for the number of dimensions. The remaining values are single-precision (32-bit) IEEE 754 floating-point numbers which comprise all of the data in row-major order; consecutive values are from the same event, not the same dimension. A Python script was written that can transform from CSV to this simple binary format and vice versa. The clustering algorithms assume any files with a `.bin` extension are binary files, and anything else it attempts to parse as CSV file.

### 2.3.3 Clustering

After the preprocessing of the data set is complete, clustering begins. A one-to-many relationship exists between a particular data set and the clustering analyses that can be performed on it. There are two main sources of different clustering analyses. First, no single clustering algorithm has been shown to be universally superior to all others for all purposes, and different algorithms can be expected to give different clusters and different numbers of clusters. Depending on the specific objectives of the analysis, different algorithms may be chosen. For example, one algorithm may be good at exploratory analysis of the data, able to locate very small clusters out of the entire data set, while another may be good at consistently profiling larger clusters — which is useful for inference across data sets. While this thesis focuses on only two clustering algorithms, many more may be used.

Secondly, for any particular clustering algorithm it may be desirable to do parameter sweeps (such as varying the number of starting clusters, or the parameters used by information criteria such as MDL) as well as sensitivity analyses. Using many different algorithms, each with many different starting parameters, results in an analysis with dozens or even hundreds of clusterings. Finally, intuitive visualization of the results is necessary in order for experts to gauge the quality of the results and to determine the biological implications of the results (i.e. to what cells do the clusters correspond).

A very important, yet under-developed, aspect of flow cytometry research is good statistical inference. In short, biologists are interested in the change in various cell populations (clusters)
across samples. Here the one-to-many relationship between a particular data set and the clustering analyses is expanded to a many-to-many relationship with multiple data sets, each with various clusterings. As previously mentioned, even a single clustering algorithm can be slow on a single CPU. Clearly there is a great need to have accelerated algorithms in order for the workflow to be feasible in reasonable time.

### 2.4 GPGPU

Traditional von Neumann general purpose processor architectures support only a single active thread of execution. Sequencing of operations is quite easy with this architecture — however concurrency is difficult. While the capabilities of general purpose processors to exploit parallelism have improved significantly throughout the years due to techniques such as the Tomasulo algorithm, superscalar dispatching, vector instructions, symmetric multi-threading, and multiple-core technologies, they still remain limited to a relatively small number of concurrent threads. In addition, a significant portion of on-chip resources are dedicated to decoding massive instruction sets for general purpose computing, branching, synchronization, pipelining, and caching. The resources available for raw floating point computations (used heavily in clustering) are comparatively limited. Effectively meeting the computing requirements of scientific applications using general purpose processors often requires hundreds if not thousands of processors.

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 [7] shown in Figure 2.5. 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 general purpose threads. This is a massively parallel architecture with many benefits over a general purpose desktop processor for data-intensive computations such as data clustering.

The current generation of the NVIDIA Tesla GPU architecture (GT200) contains upwards of 240 concurrent processing elements [7]. NVIDIA’s next generation architecture code-named Fermi is due for release in the first half of 2010 and has 512 cores [28]. 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 flow cytometry data clustering, whose computation is composed almost entirely of floating-point operations. NVIDIA’s CUDA Zone website boasts a wide variety of conference papers, articles, and project websites with application speedups upwards of 300x on graphics cards costing much less than a typical desktop computer [13].

There are a variety of different approaches to GPGPU. As previously mentioned, the OpenGL Shader Language can be used to write general purpose procedures. Another significant effort was the Brook GPU language and toolset by Ian Buck et al. at Stanford University [29]. Brook GPU uses a modified C language with extensions for stream processing. One benefit of the Brook GPU project is that the compiler can produce target code for a wide variety of different graphics cards from ATI, Nvidia, and Intel. The Brook GPU project is no longer under active development. Other emerging technology for GPGPU include OpenCL, PGI, and Microsoft’s DirectX DirectCompute. NVIDIA’s CUDA framework and hardware based on the Tesla architecture was chosen for this thesis because it is currently the most mature tool chain and best hardware for GPGPU. There is a significant amount of both commercial and development community support. Also the majority of
GPGPU research in recent years has been using CUDA technology.

### 2.4.1 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.

Host code is typically written in standard ANSI C/C++ (although CUDA has introduced Fortran support in recent versions) and compiled with a standard C compiler such as the GNU C Compiler (gcc/g++). Anything inside CUDA (.cu) files is compiled with NVIDIA’s nvcc compiler. The nvcc compiler acts as a wrapper that will compile any host code found within CUDA (.cu) files with the host compiler (typically gcc in unix environments and Microsoft’s C compiler in Windows environments). Device functions are compiled into PTX code, which is a pseudo-assembly language that is common to all NVIDIA CUDA hardware, and finally to PTX bytecode. The CUDA driver is responsible for compiling the PTX code into machine instructions for the specific GPU architecture on the system. This compilation is done on the fly (JIT compiling) and allows the result of the compiled CUDA program to be portable to different machines with different GPUs. CUDA hardware is therefore compatible with other technologies such as OpenCL as long as they can generate PTX code. More details on PTX can be found in NVIDIA’s reference materials [30]. Figure 2.6 summarizes the different layers of the CUDA architecture.

While instruction set architectures for general purpose CPUs seek to abstract the details of the hardware from the software as much as possible, knowledge of the underlying hardware is still important for programmers of performance critical applications. Programming for CUDA is very “close to metal”, and as such requires a thorough understanding of the underlying parallel architecture, thread model, and memory model to create effective implementations on the GPU.

### Hardware Architecture

At heart of the Tesla Architecture [7] are streaming multiprocessors (SMs). Each SM is comparable to a CPU with an interface for fetching instructions, decoding instructions, functional units, 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. Figure 2.7 shows the scalable model of the graphics hardware from a CUDA perspective.

**Thread Model**

The programming model and use of threads is best explained by the CUDA Programming Guide [9]. Since understanding the thread model is essential to effective programming with CUDA and understanding CUDA program implementations, this section provides a brief overview.

The massively parallel architecture with many cores requires a robust thread organization structure. 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.
Figure 2.7: CUDA Multiprocessor Architecture [9]

Figure 2.8: CUDA Thread Hierarchy [9]
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). Figure 2.8 shows an example of a kernel grid with six blocks and a two-dimensional block of threads. 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.

Internally each warp is broken again into half-warsps of 16 threads. Whenever a branch, uncoalesced global memory access, or shared memory bank conflict occurs within a half-warp, the threads diverge and must be executed serially (resulting in lower performance). This division allows programs with SIMD parallelism to have high efficiency while still allowing the program to be flexible with separate threads performing different instructions when necessary. It also avoids exposing the width of the vector hardware to the software; therefore making kernels scalable to new architectures without changing the code. NVIDIA calls this architecture SIMT — single-instruction multiple-thread.

Blocks can be compared to independent tasks (processes) running on an operating system such as Windows or Linux. All block resources are statically defined rather than being swapped out when other blocks are running — this essentially eliminates task switching overhead. When one block is paused or stalled (such as waiting for a global memory access), another active block can be executing on the multiprocessor. The number of active blocks on a multiprocessor depends on the resource requirements of the kernel. Fewer resources per thread block allow blocks more to be active. There is a maximum of 8 blocks, 32 warps, or a total of 1024 threads active per SM — whichever comes first. The ratio of the number of active warps possible for a kernel to the maximum allowed by the hardware is a metric called occupancy. Occupancy will be discussed in more detail in the Results chapter.

Memory Model

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 [9]. Memory is divided into three main categories: registers, shared memory, and global memory. Figure 2.9 summarizes which memories are accessible by threads, thread blocks, and kernel grids.

The registers are high-speed memory. There are 8192 32-bit registers per multiprocessor on the
Figure 2.9: CUDA Memory Model [9]
8800 series graphics cards (16384 in newer cards, like the GTX 200 series) and they are divided evenly among the threads in the block. Thus for a block with 512 threads, each thread can have a maximum of only 16 registers. If the program requires more registers, then the compiler assigns global memory as local memory for the thread. Threads cannot access registers reserved for other threads, regardless of whether or not the threads are physically executing at the same time. If there are not enough registers for a thread’s local memory it spills over into global memory.

Shared memory is high-speed on-chip memory and is organized into 16 banks, totaling 16 KB per multiprocessor. All threads within a block can access the shared memory; however threads must be synchronized to avoid race conditions and to guarantee that the expected value has actually been written to shared memory. Multiple threads in a half-warp attempting to access the same bank causes a bank conflict, and the accesses are serialized. The exception to this rule is if all threads in a half-warp are reading the same element; then there is a broadcast mechanism to deliver that value to all threads without a conflict. Devices with compute capability 1.1 or higher support atomic operations for writing to memory; however in the interest of compatibility with all CUDA capable devices this thesis does not utilize those functions.

The global memory space is larger than shared memory (almost 1 GB on the GTX 260), but significantly slower with delays up to hundreds of cycles. All threads in the grid can access the global memory, and it persists throughout the lifetime of the application. Thus global memory must be used when sharing data between thread blocks and between different kernel launches. Additionally there are two read-only memories, constant and texture. Both of these memories reside in the global DRAM but are cached in higher speed memory on the multiprocessor, and thus can provide higher performance than global memory if access patterns exhibit spatial locality.

Global memory accesses on the GPU have a delay of hundreds of cycles. It is very important to structure the data on the GPU such that the number of global memory transactions is reduced to improve the performance of the algorithm. Unlike CPUs which tend to have relatively narrow memory buses but a sophisticated caching hierarchy, modern GPUs have wide memory buses upwards of 512 bits wide, but with very limited caching (just constant and texture memory, and a small block of shared memory for each multiprocessor). If the threads within a thread block of a GPU kernel are accessing consecutive elements from memory at the same time, then multiple elements from a single global memory transaction can be used. This is called memory coalescing.

Older devices with compute capability 1.0 or 1.1 have strict requirements for memory coalescing. All threads within a half-warp (16 threads) must be accessing consecutive elements from memory, in order, and aligned to an address equal to 16 times the size of the elements being accessed. If any of these conditions are not met, separate transactions are issued for each thread. Newer devices with compute capability 1.2 or 1.3 have more relaxed restrictions. Threads can access items out of order. The starting address can be misaligned — this will result in 2 memory transactions, but much better than 16. Newer devices do not make a distinction between coalesced and uncoalesced
accesses — it is the total number of required transactions that is important. Fortunately an algorithm that coalesces on an older device will also be optimal for a newer device.

This chapter began with an introduction to data clustering and the different types of clustering implemented in this thesis. There are a myriad of fields, both academic and commercial, which utilize data clustering. One such field called flow cytometry was introduced. The de-facto analysis used by the flow cytometry community is manual gating, which is considered by many to be a tedious and often inaccurate (or at least incomplete) analysis technique. There has been a surge in the flow cytometry research community in recent years to use multivariate data clustering to analyze flow cytometry data. The size of the data sets and the complexity of clustering algorithms on such data sets make analysis on a CPU cumbersome and slow. Using GPUs as a co-processor it is possible to accelerate clustering methods such that processing of flow cytometry data or data from other fields is significantly faster. The next chapter presents details about the clustering algorithms accelerated in this thesis and discusses previous GPGPU efforts. Chapter 4 then describes the parallelization strategy for each algorithm on GPUs and extends it to multiple nodes in a high performance computing cluster.
Chapter 3

Supporting Work

This chapter discusses the specific algorithms being targeted for implementation using CUDA. Prior work with data clustering algorithms using GPUs is also discussed.

3.1 Algorithms

The work for this thesis focuses on two algorithms. The first algorithm is C-means — a soft or fuzzy implementation of the popular $k$-means algorithm. The second algorithm is expectation maximization (EM) with Gaussian mixture models combined with top-down agglomerative hierarchical clustering.

3.1.1 C Means

C-means is a soft, or fuzzy, version of the $k$-means least-squares clustering algorithm. Rather than every data point’s being associated with only the nearest cluster center (where nearest in this case means the smallest Euclidean distance), data points have a membership ranging from 0 to 1 in every cluster.

The algorithm is based on the minimization of the total error associated with a solution as defined in Equation (3.1) [12]. It is the sum of the squared distances of each data point to each cluster center, weighted by the membership of the data point to each cluster, for all data points.

$$E = \sum_{i=1}^{N} \sum_{j=1}^{M} u_{ij}^p \| x_i - c_j \|^2$$  \hspace{1cm} (3.1)

In Equation (3.1), $p$ 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. A fuzziness of $p = 1$ is equivalent to hard clustering and as $p \rightarrow \infty$ the membership in all clusters is equal. The fuzzy clustering is done through an iterative optimization of Equation (3.1). Each iteration, the membership $u_{ij}$ is updated using Equation (3.2), and the cluster centers $c_j$ are updated using Equation (3.3).
\[ u_{ij} = \frac{1}{\sum_{m=1}^{M} \left( \frac{\|x_i - c_j\|}{\|x_i - c_m\|} \right)^{2-p}} \]  

(3.2)

\[ c_j = \frac{\sum_{i=1}^{N} u_{ij}^p \cdot x_i}{\sum_{i=1}^{N} u_{ij}^p} \]  

(3.3)

The following is an outline of the fuzzy c-means algorithm.

1. Given the number of clusters, \( M \), randomly choose \( M \) data points as cluster centers.

2. Compute the membership value of every data point for each cluster.

3. For each cluster, sum the data points weighted by their membership in that cluster.

4. Recompute each cluster center by dividing by the total membership of the cluster.

5. Stop if there is minimal change in the cluster center; otherwise return to 2.


### 3.1.2 Gaussian Mixture Models

Data in flow cytometry is composed of many possibly overlapping clusters. The data for each vector (or event) is an aggregate of a mixture of multiple distinct behaviors. Mixture distributions form probabilistic models composed of a number of component subclasses (clusters) \[10\]. Given a \( D \) dimensional data set, each cluster \( m \) is characterized by the following parameters \[10\].

\[ \pi_m : \text{the probability that a sample in the data set belongs to the subclass} \]

\[ \mu_m : \text{a } D \text{ dimensional mean} \]

\[ R_m : \text{a } D \times D \text{ spectral covariance matrix} \]

Assuming there are \( N \) flow cytometry events \( Y_1, Y_2, \cdots, Y_N \), then the probability that an event \( Y_i \) belongs to a Gaussian distribution is given by the following Equation \[10\].

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

(3.4)
It is not known to what subclass each event belongs. Therefore it is necessary to calculate the likelihood for each subclass and apply conditional probability [10].

\[ p(y_n|\theta) = \sum_{m=1}^{M} p(y_n|m, \theta) \pi_m \]

Neither the statistical parameters of the Gaussian Mixture Model, \( \theta = (\pi, \mu, R) \), nor the membership of events to subclasses are known a priori. An algorithm must be employed to deal with this lack of information.

**EM**

Expectation maximization is a statistical method for performing likelihood estimation with incomplete data [12]. The objective of the algorithm is to estimate \( \theta \), the parameters for each subclass. First each event \( y_n \) is classified based on the likelihood criteria in Equation (3.4). 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 subclass. The membership value is the ratio of the weighted likelihood to the total weighted likelihood of all subclasses — see Equation (3.5).

\[ p(m|y_n, \theta) = \frac{p_{y_n|x_n}(y_n|m, \theta)\pi_m}{\sum_{l=1}^{M} p(y_n|l, \theta)\pi_l} \quad (3.5) \]

The subclass parameters, \( \theta \), are re-estimated based upon the new membership values completed in the E-step [10]. The event classification (E-step) and re-estimation of subclass parameters (M-step) repeats until the change in likelihoods for the events is less than some \( \epsilon \).

\[ \bar{N}_m = \sum_{n=1}^{N} p(m|y_n, \theta) \quad (3.6) \]
\[ \bar{\pi}_m = \frac{\bar{N}_m}{N} \quad (3.7) \]
\[ \bar{\mu}_m = \frac{1}{\bar{N}_m} \sum_{n=1}^{N} y_n p(m|y_n, \theta) \quad (3.8) \]
\[ \bar{R}_m = \frac{1}{\bar{N}_m} \sum_{n=1}^{N} (y_n - \bar{\mu}_m)(y_n - \bar{\mu}_m)^t p(m|y_n, \theta) \quad (3.9) \]
Hierarchical Clustering Stage

The clustering begins with a user-specified number of clusters. The algorithm then performs EM on the clusters and determines the Gaussian model parameters for each cluster. This involves computing Equation (3.5) for every event for every cluster and then Equations (3.6), (3.7), (3.8), and (3.9) for each cluster. The objective of this portion of the algorithm is to estimate the proper number of models to describe the data using an information criterion.

A Minimum Description Length (MDL) score [31], (often called a Rissanen score), is then calculated using Equation (3.10). The Minimum Description Length (MDL) principle extends the classical maximum likelihood principle by attempting to describe the data with the minimum number of binary data required to represent the data with some precision [31]. Likelihood maximization has an inherent flaw that it does not protect against over-fitting. The MDL score, in contrast, places greater weight on describing the data in fewer models. The score in equation (3.10) is computed for each configuration in the top-down agglomerative clustering to try to determine the optimal number of clusters.

\[
MDL(m, \theta) = - \sum_{n=1}^{N} \log \left( \sum_{m=1}^{M} p(y_n|m, \theta) \pi_m \right) + \frac{1}{2} L \log(ND) \tag{3.10}
\]

\(L\) is the number of continuously valued real numbers required to specify the parameter \(\theta\) [10]. \(D\) is the number of dimensions, \(N\) is the number of vectors in the input, and \(M\) is the number of clusters in the mixture.

\[L = M \left( 1 + D + \frac{(D+1)D}{2} \right) - 1 \tag{3.11}\]

The algorithm then attempts to combine the two most similar clusters. In this case, similarity is based upon the Gaussian model parameters. A distance function is computed between all possible combinations of clusters. The two clusters with the minimum distance (i.e. most similar) are combined into a new cluster. The distance function is defined by [10] as:

\[
d(a,b) = \frac{N \bar{\pi}_a}{2} \log \left( \frac{|R_{(a,b)}|}{|R_a|} \right) + \frac{N \bar{\pi}_b}{2} \log \left( \frac{|R_{(a,b)}|}{|R_b|} \right)
\]

This process repeats until the data have all been combined into a single cluster. Finally, the configuration with the minimum Rissanen score is output as the optimal solution. The results are two-fold. First, there are the statistical parameters, \(\theta = (\pi, \mu, R)\), for each Gaussian cluster. Second, all events have membership values for every cluster. Figure 3.1 summarizes the basic steps of the clustering procedure.
3.2 Data Clustering on GPGPU

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 [32]. The implementation in [32] 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 [33], 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 nonspherical 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 thesis 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 [34]. 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 [2]. Shalom et al. implemented C-means on an NVIDIA 8800 GTX using the OpenGL Shader Language with speedup of 94x [35]. 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 [1]. This thesis 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 [35] 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 thesis implemented an Expectation Maximization (EM) algorithm with Gaussian Mixture Models (GMMs). A recent publication from Kumar et al. [3] 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 [36] and have achieved overall speedup factors of 20x, and over 100x for specific portions of the algorithm.

Chapter 2 has provided an overview of data clustering, flow cytometry, and CUDA. This chapter has given detailed descriptions of the clustering algorithms focused on in this thesis. Both C-means and EM with Gaussians are two very popular algorithms, and as such there have been previous efforts to parallelize them using GPU co-processors. In both cases the algorithms have limitations, such as not supporting full diagonal covariance in the Kumar et al. EM implementation, and there is room for improvement in performance. The next chapter discusses the parallel implementation of these algorithms on the GPU and in what ways it improves on existing work. These algorithms are further parallelized beyond a single GPU using OpenMP and MPI to utilize multiple nodes and multiple GPUs in a high performance computing environment — which has not been done before with these algorithms.
Chapter 4

Parallel Algorithm Implementations

This chapter 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, the data model, and the kernel details are discussed for each algorithm. Pseudocode is presented for all of the kernels as well as some host-side code. 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.

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.

Throughout the chapter: $N$ is 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.

4.1 Parallel Implementation Architecture

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 cluster. OpenMP allows multiple threads to communicate via shared memory and maps well to SMP environments. Finally, GPU co-processors with the CUDA toolkit and driver accelerate the calculations done by each thread.

![Figure 4.1: Process Hierarchy](image)

Figure 4.1 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 thesis 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.

Figures 4.2 shows the multi-level MapReduce structure used for communication between the processes and threads shown in Figure 4.1. 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 4.2. 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.

![Communication and Reduction Hierarchy](image)

Figure 4.2: Communication and Reduction Hierarchy
4.2 C-means

The C-means algorithm was described in detail in section 3.1.1. In short, it iteratively optimizes the cluster centers by updating the cluster memberships with equation (4.1) and then computing new cluster centers using equation (4.2). The computation is broken into four different CUDA kernels — DistanceMatrix, MembershipMatrix, UpdateCenters, and ClusterSizes.

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

(4.1)

\[ c_j = \frac{\sum_{i=1}^{N} u_{ij}^p \cdot x_i}{\sum_{i=1}^{N} u_{ij}^p} \]  

(4.2)

There is a lot of inherent parallelism in the C-means algorithm. The most obvious is that the distance calculation between every data point and each cluster center is independent. This produces \( N \times M \) independent distance calculations for every iteration of the C-means algorithm. Each distance calculation itself requires multiple independent operations since the data set is multidimensional; however the distance measures require some form of reduction (the square root of the sum of squares for Euclidean distance) and thus are not truly independent. C-means has a good mix of coarse and fine grained parallelism. At a coarse-grained level the data points are divided among the different GPUs in the system. Within a single GPU, different clusters, dimensions, and ranges of events can be mapped to separate thread blocks. Finally individual distance and membership values can be computed by fine-grained SIMD threads.

Distributing data points to different GPUs allows the computation to scale well to either a small or large number of GPUs provided the number of data points per GPU is still large enough to benefit from GPU acceleration. The overall algorithm has speedup using a single GPU with data sets as small as 1,000 data points — refer to the results chapter for more details on speedup with various data sets. Typical flow cytometry data sets have at least 250,000 events; therefore there is no shortage of data to distribute to different GPUs. However, the speedup is best when the number of events per GPU is large (about 32,000 or more).

In order for the GPU resources to be utilized fully for all kernels, the number of clusters must be at least as large as the number of multiprocessors on the GPU (typically 16-30). Data sets can have dozens, if not hundreds, of potential clusters. Distributing the workload between GPUs via the clusters rather than the events would reduce the need for an aggregation step by the host to combine the partial cluster center results from each GPU. However, given the expected number of clusters
for flow cytometry, this distribution would not scale well to more than a few GPUs without causing each GPU to become underutilized. Thus the more scalable solution is to distribute data points to different GPUs.

The following is an overview of the different stages of the multi-GPU C-means implementation. The GPU kernels will be described in detail with pseudo-code later in this section.

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

Figure 4.3 is pseudo-code for the C-means algorithm on the host. OpenMP uses pre-processor pragma statements to denote parallel sections. All code inside the \#omp parallel block is executed by multiple threads (one per GPU). The \#omp barrier synchronizes all threads, similar to an MPI_Barrier or a CUDA syncthreads function call. Tasks outside the parallel section, or within a \#omp master statement, are executed only by the master thread. Operations that are common to both threads and that do not require a CUDA context, (e.g reading the input file, declaring memory structures on the host, and outputting results), are placed outside the parallel block.
MPI_Init_Thread();
rank = MPI_Comm_rank(MPI_COMM_WORLD);
data = readInputData()
initializeHostMemory()
clusters = seedClusters()
G = cudaGetDeviceCount()
omp_set_num_threads(num_gpus)
B_D = N/G/NUM_THREADS_DISTANCE;
B_M = N/G/NUM_THREADS_MEMBERSHIP;
B_C = M/NUM_CLUSTERS_PER_BLOCK;
#pragma omp parallel shared(centers,change,memberships)
tid = omp_get_thread_num()
cudaSetDevice(thread_id)
cudaHostToDevice(data)
cudaHostToDevice(clusters)
while change > epsilon
    DistanceMatrix<<< dim3(B_D,M), NUM_THREADS_DISTANCE>>>()
    MembershipMatrix<<< dim3(B_M,M), NUM_THREADS_MEMBERSHIP>>>()
    UpdateCenters<<< dim3(B_C,D), NUM_THREADS_CENTERS>>>()
    ComputeSizes<<< M, NUM_THREADS_SIZES>>>()
    centers[tid] = cudaDeviceToHost()
    sizes[tid] = cudaDeviceToHost()
    #pragma omp barrier
    #pragma omp master
        centers[MASTER] = reduceCenters(centers)
        sizes[MASTER] = reduceSizes(sizes)
        MPI_Reduce(centers[MASTER],M*D)
        MPI_Reduce(sizes[MASTER],M)
        MPI_BARRIER()
        if rank == 0
            centers[MASTER] = centers[MASTER] / sizes[MASTER]
            MPI_Broadcast(centers)
            change = computeChange()
            #pragma omp barrier
        centers[tid] = centers[MASTER]
cudaHostToDevice(clusters)
    DistanceMatrix<<< dim3(B_D,M), NUM_THREADS_DISTANCE>>>()
    MembershipMatrix<<< dim3(B_M,M), NUM_THREADS_MEMBERSHIP>>>()
    memberships = cudaDeviceToHost()
MPI_Gather(memberships)
if rank == 0:
    outputResults(centers,memberships)
MPI_Finalize()
4.2.1 Data Organization

The input data are read from either an ASCII CSV or binary file and stored as a one-dimensional array of 32-bit floating point numbers. The data set is far too large for shared memory on the device; therefore it must be kept in the device’s global memory. Concurrent threads within the kernels access the same dimension of consecutive data points, as opposed to the various dimensions of the same data point; therefore the data set is stored as a $D \times N$ matrix rather than a $N \times D$ matrix. This arrangement allows for coalesced memory accesses. The previous work in [1] did not access all event data in a fully coalesced access pattern, which hurt the performance considerably, particularly on older 1.0 compute capability devices, which have much stricter memory coalescing rules.

On the host the input data set is marked as shared memory between all of the OpenMP host threads. Each host thread copies the input data to the global memory of its corresponding CUDA device before clustering begins.

In order to facilitate the aggregation of results from multiple GPUs, the host allocates an area of shared memory for the partial cluster centers from each GPU for a total of $G \times M \times D$ floats. Each GPU has its own copy of the cluster centers ($M \times D$ floats) in global memory. Since the cluster centers are used repeatedly (once for each distance computation) they are cached in shared memory on the GPU at the beginning of the DistanceMatrix kernel.

In addition to the input data, the device also stores a matrix of all membership values (for a single iteration). In the OpenMP and MPI implementations each GPU needs to store only memberships relevant to its share of the input values. Therefore the GPU requires a total of $4(D + M)(\frac{N}{G})$ bytes of storage for the input data and memberships. A typical flow cytometry file with 24 dimensions, 500K events, and 100 clusters requires 236 MB of memory with a single GPU.

4.2.2 Distance Kernel

The first kernel computes the distance between each data point and all of the cluster centers. It has a computational complexity of $O(NMD)$. The distance kernel is launched with a two dimensional grid of $N/G/512 \times M$ blocks. It computes an $M \times N/G$ matrix where element $d_{ij}$ is the Euclidean distance between the $j^{th}$ event and the $i^{th}$ cluster center. Each thread block in the grid computes 512 elements (one per thread) of the matrix. The matrix is stored in global memory and is used by the Membership kernel.

The code for the kernel can be found in Figure 4.4. $\text{blockIdx.x}$ and $\text{blockIdx.y}$ are kernel variables for the indices of the thread blocks in the grid. The indices correspond to the events range and the cluster number, respectively. Each thread within the block is executing the kernel in parallel. The only difference for each thread is the value of the thread index: $\text{tid}$. Consecutive threads access consecutive elements of the data array; therefore all global memory reads are coalesced on devices
of any compute capability. A small error of $10^{-30}$ is added to the distance to prevent potential divide by zero errors (NaN) in the membership calculations when the data point is exactly equal to the cluster center. An alternative solution is to use a conditional check for a near-zero value in the membership calculation, but branches are bad for CUDA performance because the warps can diverge resulting in serial execution.

```c
void distanceMatrix(float* clusters, float* data, float* distances)
{
    int c = blockIdx.y; // cluster number
    // copy cluster center into shared memory
    __shared__ float center[D];
    for(i = tid; i < D; += NUM_THREADS)
        centers[i] = clusters[c*D+i]
    __syncthreads();

    // compute Euclidean distance for each event
    int i = blockIdx.x * blockDim.x + tid;
    if(i < N) {
        distance = 0;
        for(d = 0; d < D; d++)
            distance += pow(data[d*N+i]-center[d],2);
        distances[c * N + i] = sqrt(distance+1e-30);
    }
}
```

Figure 4.4: Pseudo-code for Distance Kernel

Grid size considerations

An earlier implementation attempt used a grid of $M$ blocks (one per cluster). Threads iterated over the events in a for loop and each block computed one row of the distance matrix. This reduced memory accesses slightly since each cluster center was only loaded into shared memory once. Despite fewer global memory accesses, the $M$ block implementation performed poorer with a small to moderate number of clusters. Figure 4.5 compares the performance of the two implementations. The device’s computing resources are under-utilized when the number of blocks is not an even multiple of the number of SMs on the card. The test was done on a C870 device with 16 SMs; therefore the two curves are closest at 16, 32, and 48 clusters. Before 16 clusters, not even all of the SMs are being used. Even at 16 clusters, there is only one active block per SM (25% occupancy), so the computing resources are often idling while all active warps are stalled waiting for global memory accesses or register dependencies. At 17, a second batch of blocks must be launched resulting in a spike in execution time. Finally at 32 clusters, the $M$ grid has good occupancy (50%) so the run-time is nearly identical to the $N/G/256 \times M$ grid. Having many more blocks that each have a smaller workload is helpful for good occupancy and full use of device resources. Overhead from the
for loop was also eliminated — the threads perform their tasks based on the grid and thread indices provided by the scheduling hardware rather than maintaining a separate loop in software. It also provides some longevity to the implementation since the number of multiprocessors is expected to continue increasing on future devices; implementations with a shortage of blocks cannot be expected to scale to newer hardware.

![Figure 4.5: Effect of Grid Size on Distance and Membership Kernels](image)

**4.2.3 Membership kernel**

The second kernel generates an $M \times N$ matrix of membership values using the distance matrix computed in the previous kernel. It computes Equation (4.1) and raises each value to the power of $p$. Three different implementation strategies were attempted for the membership matrix.

**Direct implementation**

The first membership kernel directly implements the update equation (4.1). Thus it independently computes the membership of every data point for every cluster. This technique is $O(NM^2)$. Computing the membership for each cluster requires the sum of the distances for all $M$; thus it is quadratic
with respect to the number of clusters. All distances have already been computed; therefore the di-

dimensionality of the data does not affect the membership kernel.

Similar to the distance kernel, it is launched with a grid of \( N/G/256 \times M \) blocks, where 256 is

the number of threads per block. Each block computes up to 256 elements of one row of the \( M \times N \)

matrix. Refer to Figure 4.6 for the kernel code.

---

```c
float membershipValue(float* distances, int c, int e) {
    myDist = distances[c*NUM_EVENTS+e];
    denom = 0;
    for(int i = 0; i < NUM_CLUSTERS; i++) {
        otherDist = distances[i*NUM_EVENTS+e]
        denom += pow(myDist/otherDist,2/(FUZZINESS-1))
    }
    return 1/denom;
}

void membershipMatrix(float* distances, float* memberships, int N) {
    int c = blockIdx.y; // cluster number
    int e = blockIdx.x * blockDim.x + tid; // event number
    if(e < N) {
        membership = membershipValue(distances,c,e);
        memberships[c*N + e] = pow(membership,FUZZINESS);
    }
}
```

---

Figure 4.6: Pseudo-code for Membership Kernel

**Partition Camping**

Global memory on the GPU is divided into different partitions (banks). Partition camping is the

term NVIDIA has coined for an uneven distribution of memory requests to the partitions [37]. The

number of 256-byte wide memory partitions varies depending upon memory-bus width of the GPU.

The channels are 64-bits wide; therefore 384-bit buses have six, 448-bit buses have seven, and

512-bit buses have eight. The partition camping problem is difficult to detect since it can depend

heavily on the memory bus width, number of SMs, kernel grid size, thread block size, occupancy,

and the dimensions of the data being accessed. The problem was first noticed on a Tesla C1060

GPU with the *Distance* and *Membership* kernels when using an \( M \times N/256 \) kernel grid. Blocks

are scheduled based on their one dimensional block id as in Equation 4.3 [37].

\[
bid = blockIdx.x + gridDim.x * blockIdx.y 
\]  

(4.3)

Blocks traversed the \( D \times N \) data and \( M \times N \) distance matrices in a column-major order. Consecutive blocks accessed the same 256 columns (1024 bytes) of the distance matrix. Therefore, in the worst case, memory accesses would queue at only 4 of the 8 partitions, and the others would
be idle. This resulted in a performance degradation of about 2x for certain numbers of clusters and input sizes. Figure 4.7 shows an example of partition camping impacting performance on the C1060 GPU. Once the number of clusters (rows of the matrix) exceeded the number of blocks that could be scheduled to the GPU, the majority of the blocks would all be trying to access only 4 of the 8 partitions of memory.

The solution to partition camping for these kernels was to transpose the dimensions of the grid: \( N/256 \times M \) instead of \( M \times N/256 \). Consecutive blocks then traverse the distance matrix in a row-major order and memory accesses are distributed to all memory partitions fairly evenly.

\[
\begin{array}{c|c|c}
\text{Number of Clusters} & \text{Column-major block scheduling} & \text{Row-major block scheduling} \\
\hline
50 & \text{\textcolor{red}{4.5}} & 4.0 \\
100 & 1.5 & \text{\textcolor{red}{4.0}} \\
150 & 1.0 & \text{\textcolor{red}{3.5}} \\
200 & 0.5 & \text{\textcolor{red}{3.0}} \\
250 & 0.5 & \text{\textcolor{red}{2.5}} \\
300 & \text{\textcolor{red}{1.0}} & \text{\textcolor{red}{2.0}} \\
\end{array}
\]

Figure 4.7: C-means Partition Camping

**Linear Membership Kernel**

Directly implementing the membership update equations as in the previous section, albeit simple, is not very efficient. The denominator in the sum of the membership equation is the same for all clusters — thus it needs to be computed only a single time per event rather than \( M \) times. This reduces the computation complexity of the membership matrix calculation to \( O(NM) \) [38]. Equation (4.1) can be rewritten equivalently as (4.4).

\[
s_i = \frac{1}{\sum_{k=1}^{M} \left( \frac{1}{\|x_i - c_k\|^{p-1}} \right)} \quad \forall \ i \in [1, N]
\]
\[ u_{ij} = \frac{1}{(\|x_i - c_j\|)^{\frac{2}{p-1}} * s_i} \quad \forall i \in [1, N], \forall j \in [1, M] \] (4.4)

The kernel is launched with \( N/G/256 \) blocks. While this has \( M \) fewer blocks than the previous implementation, it still provides a sufficient number of blocks to utilize the GPU resources fully on reasonably sized data sets. 16K events will provide at least 50% occupancy on current generation devices. Source code for the kernel can be found in Figure 4.8. Every block in the grid computes 256 columns of the \( M \times N \) membership matrix. Each thread sums the distances to each cluster for its data point. It then loads each distance again and computes the final membership value. Although it is redundant to load each of the distances twice, loading the distances into shared memory would require \( 256 \times M \times 4 \) bytes (\( M \) floats per thread), which would limit the maximum clusters to only 256, and performance would degrade after 128 clusters due to lower occupancy. Note that the final membership values can be stored in the original distance matrix with this implementation. This effectively cuts the device memory requirement in half, since it does not need to store separate \( M \times N \) matrices for the distance and membership values.

```c
__global__ void ComputeMembershipMatrixLinear(float* distances, int N) {
    float denom = 0.0f;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < N) {
        for(int c=0; c < M; c++) {
            dist = powf(distances[c*N+i],2/(FUZZINESS-1));
            denom += 1 / dist;
        }
        for(int c=0; c < M; c++) {
            dist = powf(distances[c*N+i],2/(FUZZINESS-1));
            membershipValue = powf(dist*denom,-FUZZINESS);
            distances[c*N+i] = membershipValue;
        }
    }
}
```

Figure 4.8: Pseudo-code for Membership Kernel

### 4.2.4 Update Centers kernel

After all of the memberships have been computed, another kernel is launched that computes the new cluster centers using Equation (4.2). A variety of different parallelization strategies were investigated for this kernel. The first attempt, as in [1], recognized that the computation of every cluster center is completely independent, and thus can easily be allocated to separate blocks in a grid of size \( M \). Threads within the block sum all of the input data weighted by the membership values. The partial results from each thread are reduced to form the final numerators for each dimension.
and finally divided by the size of each cluster. This strategy has several short-comings. The most prominent problem is that it requires a large amount of shared memory if each thread stores its partial sum for all $D$ dimensions. Either the number of threads has to be reduced as $D$ increases (causing low occupancy) or significantly slower global memory must be used. There are also not enough blocks to utilize the device resources fully if the number of clusters is small.

Aside from the cluster centers being independent, every dimension of the multivariate center is independent. Therefore the kernel grid can be expanded to $M \times D$, where each block computes one dimension of one cluster center. The denominator of equation (4.2) is the same for each cluster center, but the membership value is already loaded into registers for the numerator calculation. Repeating the denominator summation $D$ times is not a significant inefficiency. Higher occupancy and the ability to scale to a much smaller or much larger number of clusters is far better.

Although the $M \times D$ 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 $M/B \times D$ 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.

Figure 4.9 is the CUDA implementation for the UpdateCenters kernel. The kernel is launched with a grid of $M/4 \times D$ blocks. Threads within the block loop through all of the events. A data element is loaded into a register for each thread. Every thread then loops over 4 membership values (from the membership matrix computed in the previous kernel) and adds its contribution to each cluster center in shared memory. After processing all of the events, the partial results are reduced by addition to a single value. Another small kernel computes the cluster sizes (denominator of equation 4.2) by reducing the $M \times N$ membership matrix to an $M$ length vector with addition.

The GPU is not able to compute the entire cluster center by itself since the work is distributed to multiple GPUs and they cannot communicate directly with one another other than through the host. Each GPU, $g$, returns both the numerators and denominators of Equation (4.2), and the host aggregates the results from all GPUs to compute the new cluster centers using Equation (4.5). $4GM(D + 1)$ bytes are transferred from the GPUs to the host for each iteration of C-means.

Reduction of the cluster centers occurs in two stages. First the results from each GPU on a
void UpdateCenters(float* memberships, float* data, float* centers, int N)
{
    __shared__ float numerators[NUM_THREADS*B];

    int d = blockIdx.y; // dimension
    int c_start = blockIdx.x*B
    int c_num = B
    if(blockIdx.x == gridDim.x-1 && M % B) // boundary condition
        c_num = M % B;

    for(int c=0; c < c_num; c++)
        numerators[c*NUM_THREADS+tid] = 0;

    for(int e = tid; e < N; e+=NUM_THREADS)
        dataVal = data[d*N+e];
        for(int c=0; c < c_num; c++)
            numerators[c*NUM_THREADS+tid] += memberships[(c+c_start)*N + e] * dataVal;

    __syncthreads();

    for(int c=0; c < c_num; c++)
        numerators[c*NUM_THREADS+tid] = parallelSum(&numerators[NUM_THREADS*c])

    if(tid == 0)
        for(int c=0; c < num_c; c++)
            centers[(c+c_start)*D+tid] = numerators[c*NUM_THREADS];
}

Figure 4.9: Pseudo-code for UpdateCenters kernel
single node are combined by the master OpenMP thread in a shared memory location. The master threads then transmit the results to the root node using MPIReduce. The root node computes the final center values and broadcasts them to each node. Each thread on the node then copies the centers from shared memory onto its GPU. The master thread compares the new cluster centers to the old cluster centers and stops if the total change in the centers is less than a user-specified epsilon value.

\[ c_j = \frac{\sum_{g=1}^{G} \text{numerator}_{jg}}{\sum_{g=1}^{G} \text{denominator}_{jg}} \quad \forall \ j \in [1,M] \quad (4.5) \]

**CUBLAS**

Careful observation of the center update computation reveals that it can be represented as a matrix multiplication. The numerator for each dimension of a cluster center is just a weighted sum of the membership values for the cluster and the input data. In other words, it is the dot product of the two vectors. Thus all of the numerators for all of the cluster centers are simply the multiplication of the data matrix with the membership matrix.

\[ \text{Numerators} = \text{Data} \times \text{Memberships}^T \quad (4.6) \]

The final centers are then formed by dividing the cluster centers by the size of their respective clusters (denominator of equation 4.2). In the multi-GPU case, each GPU performs an \([M \times \frac{N}{G}] \times [\frac{N}{G} \times D]\) multiplication, and all of the \([M \times D]\) matrices must be added together.

CUDA has GPU-accelerated implementations for many of the BLAS library routines. The SGEMM routine performs single-precision matrix multiplication in the following form, where \(A\), \(B\), and \(C\) are matrices and \(\alpha\) and \(\beta\) are scalars.

\[ C = \alpha AB + \beta C \quad (4.7) \]

The host-side implementation can be seen in Figure 4.10. The BLAS routines expect Fortran style (column-major) matrices but C arrays are row-major; therefore the transposes specified in the SGEMM call are the opposite of those shown in Equation (4.6). ComputeClusterSizes is a trivial kernel. Each thread sums its share of the elements and stores its contribution in shared memory. A \(\log_2\) butterfly reduction is then used to produce the final sum.

BLAS routines are typically known for their high computational efficiency and numerical stability. Unfortunately in this scenario the CUBLAS implementation often performs slower than the UpdateCenters kernel. This appears to be a result of the matrices’ being very rectangular rather than square. The CUBLAS SGEMM seem optimized for square matrices. CUDA has not released source code for the CUBLAS routines so there is no qualitative way to analyze how the relative
cublasSgemm('t','n',D,M,N,1.0,data,N,memberships,N,0.0,clusters,D);
ComputeClusterSizes<<< M, 512 >>>();
sizes = cudaDeviceToHost();
for(int i=0; i < M; i++) {
    for(int j=0; j < D; j++) {
        clusters[i*D+j] /= sizes[i];
    }
}

Figure 4.10: CUBLAS implementation of Cluster Centers

dimension sizes impact the performance.

A few tests were done using files with different numbers of events and dimensions and then
varying the number of clusters. Figure 4.11 shows the performance of SGEMM and the Update-
Centers kernel with two different data sizes. The SGEMM performance looks similar to a step
function. The size of the steps is proportional to the dominant dimension, N. This makes sense
since the FLOPs required for the multiplication is on the order of $2NMD$. However, rather than
execution time increasing linearly as $M$ increases, it remains essentially unchanged. This appears
to be an inefficient blocking strategy leading some of the device resources not being fully used.

Figure 4.11: CUBLAS SGEMM Performance on 2 data sets

Using the CUDA visual profiler it was possible to inspect the grid and block sizes used for
the SGEMM kernel. As the smaller dimension increases, utilization of resources and GFLOPS
increase, while execution time remains the same. With $M \leq 768$ the number of blocks in the
grid is $[1 \ to \ 24 \times 1]$, which is less than or equal to the number of SMs on the GTX 260; all of
the resources are not being used. Once $M > 768$, 25 blocks are required and the GPU needs to
compute the equivalent of two sets of blocks to complete the kernel. This is the similar to the effect
of insufficient thread blocks discussed in Section 4.2.2, except even more pronounced due to the
coarseness of the grid and lower occupancy. The blocks have 512 threads and are limited to 50% occupancy due to shared memory constraints; therefore only one block is active per SM and the throughput at 48 blocks is the same as 24 blocks.

As the number of clusters increases the CUBLAS implementation does eventually have higher performance than the UpdateCenters kernel. The intersection of the performance of the two implementations depends upon the sizes of $N$ and $D$; with smaller $N$ or larger $D$ values CUBLAS outperforms UpdateCenters at a lower number of clusters. Table 4.1 summarizes performance figures for a few different data sizes. For a typical flow cytometry file size, once $M \geq 384$ the throughput of the SGEMM implementation is faster. However, this is higher than the number of expected clusters for an FCS file. The SGEMM peaks at 80 GFLOPS and 15 GB/s for global memory throughput when $M = 768$ and $N = 64K$. Performance was nearly identical in a $D = 24, N = 512K$ data file. The same GPU can achieve over 400 GFLOPS with SGEMM for moderate sized square matrices ($N=M=K=4096+)$ so there is definitely room for improvement with a BLAS kernel more optimized for rectangular matrices. The UpdateCenters kernel operates at only 40 GFLOPS with 104 GB/sec (about 95% of the peak value for the GPU) for global memory throughput. The SGEMM makes use of shared memory, reducing its dependency on global memory, whereas UpdateCenters does not; global memory bandwidth quickly becomes a limiting factor for UpdateCenters. Clearly SGEMM is the more scalable solution as the number of clusters and dimensions rises. However, the UpdateCenters kernel performs faster below 256-384 clusters and over an order of magnitude faster for small numbers of clusters.

<table>
<thead>
<tr>
<th>Data size</th>
<th>UpdateCenters</th>
<th>SGEMM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$D$</td>
<td>$N$ (thousands)</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>768</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>384</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>768</td>
</tr>
<tr>
<td>24</td>
<td>512</td>
<td>100</td>
</tr>
<tr>
<td>24</td>
<td>512</td>
<td>256</td>
</tr>
</tbody>
</table>

### 4.2.5 Comparison to prior work

The implementation in [1] did not contain the Distance or Membership kernels — everything was in a single kernel. While this actually minimized the number of global memory accesses and the memory space requirement on the device (memberships and distances were not saved in global memory), there were a number of problems. First, the distances were needlessly recomputed for the denominator of the membership calculation in Equation (4.1) since the blocks (that were each computing
independent centers) cannot synchronize and share data. When dimensionality of the data exceeded four, caching all of the distance computations in global memory became faster than simply recomputing them in each block of the UpdateCenters kernel. Secondly, a single massive kernel required a large number of registers per thread resulting in low occupancy. The C-means algorithm has a relatively poor ratio of computation to memory access, making it memory bound. This makes kernel occupancy very important because it allows meaningful computation and memory accesses to overlap as much as possible. The implementation in [1] had a lot of uncoalesced memory accesses — which further compounded the memory-bound problem. Furthermore, it required $D$ floats of shared memory for every thread in the block to hold temporary results and $D \times M$ floats for the cluster centers from the previous iteration. As the dimensions increased, the number of threads per block had to decrease due to the small amount of shared memory on the device. Fewer threads per block reduced occupancy even further.

The Shalom et al. GLSL implementation stated that up to 77% of their execution time was dedicated to device-host memory copying — allowing large numbers of dimensions and clusters, but at the cost of a very large inefficiency [35]. They also compute the new cluster centers on the host rather than using GPU acceleration; only the membership matrix is computed on the GPU. Unlike Shalom et al. [35] the implementation in this thesis does not require membership values to be transferred to the host for every iteration. Membership values ($4MN$ bytes) are transferred to the host once at the end of the program if the user requests event-level result output.

The Anderson et al. implementation uses Cg rather than CUDA and performs C-means in a six phase pass: distance calculations, membership values, dot product of input data with membership values, reduction of numerators, reduction of denominators, and final cluster centers. The implementation in this thesis combines the dot product and reduction stages into a single kernel, avoids storing separate distance and membership matrices (thereby roughly halving the device memory requirement), does not store the weighted input data (reducing global memory access), and does not need to pack the data structures into the GPU’s 4096x4096 or 8192x8192 textures. In both cases, the wall clock time required per iteration of C-means is lower on comparable hardware with the implementation in this thesis.

4.3 Expectation Maximization

The Expectation Maximization with Gaussian mixture models (GMM) algorithm has a higher computational complexity than C-means, but fortunately also exhibits a lot of parallelism. The details of the algorithm are covered in detail in Section 3.1.2. In short, new cluster memberships are computed for each event in the E-step using Gaussian model parameters from the previous iteration, and then the parameters are updated using the new memberships in the M-step. Iteration continues until the total change in likelihood (which is computed in the E-step) is less than some user-specified epsilon.
value. After each EM phase, the two most similar clusters are merged until only a single cluster, or a user-specified number of clusters remains. An MDL score is computed for each configuration to try to determine the best solution.

EM with Gaussian mixture models has a complexity of $O(NMD^2I)$, where $I$ is the number of iterations. When the hierarchical clustering is used to combine similar Gaussians until only one cluster remains, the complexity grows to $O(N\frac{M(M+1)}{2}D^2I)$. Generally the time complexity is not that severe. Each time two Gaussians are combined, the algorithm is not starting from random cluster centers and requires significantly fewer iterations to converge. The following list outlines the 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.

4.3.1 Data organization
The data organization for GMM is far more complicated than C-means, which only needs to keep track of cluster centers. Every Gaussian model in the mixture has its own size ($n$), probability ($\pi$), D-dimensional mean ($\mu$), and $D \times D$ covariance matrix ($R$).

For CUDA implementations it is preferable to have flattened contiguous arrays of data rather than a lot of pointers and structures. This promotes coalesced reads from global memory in GPU kernels. It also simplifies copying data between the host and the device. Therefore the data for the implementation is all organized into a single structure of arrays (SoA) as seen in Figure 4.12. The
input data also need to be stored on the GPU. Due to the access patterns in the various kernels input data are stored in a $D \times N$ matrix, such that contiguous elements are from the same dimension rather than the same data point. The total memory requirement for a single GPU implementation is $M(3 + D + 2D^2 + N) + ND$ floats.

$$
\begin{array}{l}
\text{typedef struct} \\
\{ \\
\quad \text{float* } n; \quad \text{// expected # of events in cluster: [M]} \\
\quad \text{float* } \pi; \quad \text{// probability of cluster in GMM: [M]} \\
\quad \text{float* } \text{constant}; \quad \text{// Pre-computed constant for E-step: [M]} \\
\quad \text{float* } \text{means}; \quad \text{// Spectral mean for the cluster: [M*D]} \\
\quad \text{float* } R; \quad \text{// Covariance matrix: [M*D*D]} \\
\quad \text{float* } \text{Rinv}; \quad \text{// Inverse of covariance matrix: [M*D*D]} \\
\quad \text{float* } \text{memberships}; \quad \text{// Fuzzy memberships: [M*N]} \\
\} \text{clusters_t;} \\
\end{array}
$$

Figure 4.12: Gaussian mixture model data structure

Similar to the C-means implementation, the workload is distributed to the different GPUs by partitioning the events. Thus each GPU is responsible for $N/G$ events. For a fixed problem size, adding more GPUs decreases the memory requirements per GPU to $M(3 + D + 2D^2 + \frac{N}{G}) + \frac{N}{G}D$ floats. A large flow cytometry data file may have one million events, 24 dimensions, and upward of 100 clusters, which results in a total memory requirement on a single GPU of 473 MB of memory. With two GPUs, the memory per GPU decreases to 237 MB. This near linear reduction in memory requirements allows the implementation to scale to larger problem sizes by adding more GPUs to the system. Both of these values are suitable for modern middle-to-high-end desktop GPUs, which typically have 512 MB or more of onboard memory. Unfortunately the master host node of the system that gathers the results from each slave node still needs to be able hold the total amount of data (comparable to a single GPU) in system memory.

The host has some additional memory requirements beyond a single copy of the model parameters and the input data. The host needs to have a region of shared memory that holds the partial Gaussian model parameters ($n, \pi, \mu, R$) from each GPU. It is not necessary to share the membership values between the GPUs, which is the largest source of memory consumption for typical data sets (low to moderate dimensionality, and a large number of data points). Thus the memory overhead for the multi-GPU implementation on the host is $GM(3 + D + D^2)$ floats. This is quite small, being only 236 KB per GPU using the example in the previous paragraph. In a multi-node environment, this amount of data needs to be collected, reduced, and then broadcast to each node for every EM iteration.
4.3.2 Initialization

The CUDA implementation begins by initializing all of the model parameters. This is done by the host on the root node. The size, $n$, of each cluster is set to $N/M$. The means, $\mu$, are chosen from random data points in the input data. The covariance matrices, $R$, are set to an identity matrix.

Each host thread then copies its portion of the input data and the entire set of initial Gaussian model parameters to its corresponding GPU.

4.3.3 E-step

The details of the E-step are discussed in detail in Section 3.1.2. The E-step is broken into two kernels. The first computes the log-likelihood that a data point $y_i$ belongs to a cluster $m$ using Equation (4.8) and weights it by the $\pi_m$ of the cluster. All of the likelihood calculations are independent, resulting in $M \times N$ data parallel operations. $estep1$ is launched with a two-dimensional $B \times M$ grid of blocks. $B$ is a relatively arbitrary number, chosen experimentally to be 16. This allows for enough blocks to keep the GPU fully utilized even when the number of clusters is small, but not so many as to negate the benefits of caching model parameters in shared memory within the block. The first grid dimension determines the range of events. The second grid dimension determines the cluster. In total there are $M \times N$ memberships.

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

(4.8)

Source code for the $estep1$ kernel can be found in Figure 4.13. This implementation caches the parameters for the cluster in shared memory since they are used repeatedly for every likelihood calculation. The log-likelihood is computed rather than likelihood for the sake of numerical stability since the exponential function can overflow a 32-bit floating point number with an input as small as 90. The implementation supports the use of either full or diagonal covariance. Diagonal covariance reduces complexity from $O(NMD^2)$ to $O(NMD)$ but restricts the shape of the Gaussians.

The second step converts each weighted likelihood into a fuzzy probability for membership in each cluster using Equation (3.5). Since the first step computes log-likelihood rather than just likelihood, a log sum of exponentials must be used for the denominator. In order to avoid potential overflow of the exponential function, the equality in Equation (4.9) is used. The maximum value computed by the exponential function will be zero (resulting in 1.0) and small values will approach zero.

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

(4.9)

The membership calculation is independent for every event, but since there is a data reduction required across all clusters, there are only $N$ data-independent operations for the second part of the
void estep1(float* data, clusters_t* clusters, int D, int N) {
    int c = blockIdx.y;
    __shared__ float mean[NUM_DIMENSIONS];
    __shared__ float Rinv[NUM_DIMENSIONS*NUM_DIMENSIONS];

    // determine data indices based on block index
    compute_indices(N,&start_index,&end_index);

    copy_params_to_shared(clusters,mean,Rinv,&log_pi,&log_constant);
    __syncthreads();

    // Compute log-likelihood for every cluster for each event
    // L = constant*exp(-0.5*(x-mu)*Rinv*(x-mu))
    // LL = log(L) = log_constant - 0.5*(x-mu)*Rinv*(x-mu)
    for(int e=start_index; e < end_index; e += NUM_THREADS) {
        LL = 0.0f;
        #if DIAG_ONLY
            for(int i=0; i < D; i++)
                LL += (data[i*N+e]-mean[i]) * (data[i*N+e]-mean[i]) * Rinv[i*D+i];
        #else
            for(int i=0; i < D; i++)
                for(int j=0; j < D; j++)
                    LL += (data[i*N+e]-mean[i]) * (data[j*N+e]-mean[j]) * Rinv[i*D+j];
        #endif
        clusters->memberships[c*N+e] = -0.5f * LL + log_constant + log_pi;
    }
}

Figure 4.13: E-step log-likelihood kernel pseudo-code

E-step. The kernel is launched with a grid of \( B \) blocks. Source code for estep2 can be found in Figure 4.14.

4.3.4 M-step

The M-step updates the parameters for each Gaussian model based on the input data and the membership values computed in the E-step using equations (3.6), (3.8), and (3.9). The implementation is broken into three CUDA kernels, one for each equation. For the multi-GPU implementation, data must be aggregated by the host and copied back to each GPU between each of the three kernels. Figure 4.15 outlines the implementation of M-step on the host using OpenMP and MPI.

The reduce_n function sums \( n \) from each GPU for every cluster. Similarly, reduce_means and reduce_covariance add the value of \( \mu \) and \( R \) respectively from each GPU, but also divide each element by the size of the respective cluster computed in the first kernel.

As seen in Figure 4.15, the multi-GPU implementation of M-step requires six thread synchronizations, six GPU memory copies, and three collective MPI operations that are not necessary for a single-GPU implementation. However, the Gaussian model parameter data are very small compared to the size of the input data and the fuzzy memberships for typical data sets. The total communication overhead is \( 4 \times 2GM(1 + D + D^2) \) bytes per iteration for the M-step. A typical target flow
void estep2(float* data, clusters_t* clusters, int D, int M, int N, float* likelihood) {
    __shared__ float shared_likelihood[NUM_THREADS];
    shared_likelihood[tid] = 0.0;

    // determine data indices based on block index, grid size
    compute_indices(N,&start_index,&end_index);

    for(int e=start_index; e < end_index; e += NUM_THREADS) {
        // find the maximum likelihood for this event
        max = clusters->memberships[e];
        for(int c=1; c<num_clusters; c++)
            max_likelihood = fmaxf(max,clusters->memberships[c*N+e]);

        // Compute denominator (sum of weighted likelihoods)
        denominator = 0;
        for(int c=0; c<num_clusters; c++)
            denominator += expf(clusters->memberships[c*N+e]-max);
        denominator = max + logf(denominator);
        shared_likelihood[tid] += denominator;

        // Divide by denominator and undo the log
        for(int c=0; c<num_clusters; c++)
            clusters->memberships[c*N+e] = expf(clusters->memberships[c*N+e] - denominator);
    }

    __syncthreads();
    likelihood[blockIdx.x] = parallelSum(shared_likelihood);
}

Figure 4.14: E-step membership kernel pseudo-code

cytometry data set with 24 dimensions and 100 clusters would result in only 470 KB of memory
transfer overhead per GPU. The amount that must be transferred over the network and combined
with MPI is reduced even further by a factor of the number of GPUs per node.

The first kernel computes the size of each cluster by adding all of the membership values. This is
launched with a grid of \( M \) blocks — one per cluster. The threads iterate over the \( N \) events assigned
to the GPU and compute a partial sum which is stored in shared memory. The memberships are
stored as an \( M \times N \) array; therefore consecutive events are contiguous and loaded from global
memory with full coalescing with this implementation. The partial sums are then reduced into a
single result, which is stored in global memory and copied to the host. The host then aggregates the
partial results from each GPU to get the final cluster sizes. Each GPU must then copy the result from
shared memory on the host back to the device. Figure 4.16 is the CUDA code for the computation
of the cluster sizes via Equation (3.6).

The second kernel computes the D-dimensional mean for every cluster. It is launched with a
2-dimensional grid of \([M \times D]\) blocks. Each thread block is responsible for one dimension of one
cluster. The implementation is very similar to mstep_n. Figure 4.17 has the kernel’s pseudo-code.

The third M-step kernel computes the \( D \times D \) covariance matrix for every cluster. There
are \( D^2 M \) total elements for the covariance matrices, but covariance is symmetric. Thus only
\( M(D)(D+1)/2 \) elements actually need to be computed. The kernel is launched with a 2-dimensional
mstep_N<<<M, NUM_THREADS>>>(d_clusters, D, M, events_per_gpu);
clusters[tid].N = copy_from_gpu(d_clusters.N)
#pragma omp barrier
#pragma omp master
clusters[0].N = reduce_N()
MPI_Allreduce(MPI_IN_PLACE, clusters[0].N, M, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD)
#pragma omp barrier
copy_to_gpu(clusters[0].N)

mstep_means<<<dim3(M, D), NUM_THREADS>>>(d_data, d_clusters, D, M, events_per_gpu);
clusters.means[tid] = copy_from_gpu(d_clusters.means)
#pragma omp barrier
#pragma omp master
clusters[0].means = reduce_means()
MPI_Allreduce(MPI_IN_PLACE, clusters[0].means, M * D, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD)
clusters[0].means = clusters[0].means / clusters[0].N // element matrix op
#pragma omp barrier
copy_to_gpu(clusters[0].means)

mstep_covariance<<<dim3(M, D * D), NUM_THREADS>>>(d_data, d_clusters, D, M, events_per_gpu);
clusters.R[tid] = copy_from_gpu(d_clusters.R)
#pragma omp barrier
#pragma omp master
clusters[0].R = reduce_covariance()
MPI_Allreduce(MPI_IN_PLACE, clusters[0].R, M * D * D, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD)
clusters[0].R = clusters[0].R / clusters[0].N // element matrix op
}
#pragma omp barrier
copy_to_gpu(clusters[0].R)

__global__ void
mstep_n(clusters_t* clusters, int D, int M, int N)
{
    int m = blockIdx.x;
    __shared__ float sums[NUM_THREADS];
    sums[tid] = 0
    for(int n=tid; n < N; n += NUM_THREADS)
        sum[tid] += clusters->memberships[m*N+n];
__syncthreads();
    sum = parallelSum(temp_sums);
    if(tid == 0)
        clusters->N[m] = sum;
}

Figure 4.15: M-step host pseudo-code

Figure 4.16: M-step n pseudo-code
__global__ void mstep_means(float* data, clusters_t* clusters, int D, int M, int N)
{
    int m = blockIdx.x;
    int d = blockIdx.y;
    __shared__ float sums[NUM_THREADS];
    sums[tid] = 0
    for(int n=tid; n < N; n += NUM_THREADS)
        sum[tid] += data[d*N+n] * clusters->memberships[m*N+n];
    __syncthreads();
    sum = parallelSum(temp_sums);
    if(tid == 0)
        clusters->means[m*D+d] = sum;
}

Figure 4.17: M-step $\mu$ pseudo-code

grid of $[M/B \times D(D+1)/2]$ blocks. $B$ is a loop unrolling factor which determines how many clusters are computed per block. It attempts to strike a balance between the number of times the input data are iterated over and the shared memory/register limitations available for storing partial results from each thread inside the block. CUDA does not support 3-dimensional grids. Therefore each thread block must compute the row and column of the covariance matrix based upon the $\text{blockIdx.y}$ index.

Pseudo-code for the covariance kernel is in Figure 4.18. The actual code is manually loop unrolled by a factor of $B$ (set to six), but not shown in the Figure for brevity. This allows the data values to be loaded once into a register and then used repeatedly for 6 multiplications with cluster membership values. An unrolling factor of six keeps the register count below 32 (avoiding spillover into local memory) and keeps shared memory below 8K so that that two blocks can be active in each SM in 1.2 or greater devices. In the multi-GPU implementation the kernel computes total variance for each element rather than the covariance. The host is responsible for dividing the variance by the cluster size after it collects the partial results from each GPU. This is necessary because the events are split between multiple GPUs.

The covariance matrices are the most demanding computation in this implementation of EM with Gaussian mixture model algorithms when dealing with flow cytometry data sets. There are $N \times M \times D^2$ independent operations that can be performed in parallel. Unfortunately, doing them all in parallel requires a significant amount of redundant memory access — each membership value is accessed $D^2$ times and each data element is accessed $M \times D$ times. Since it is one of the performance bottlenecks, a number of different strategies were attempted before arriving at the implementation in Figure 4.18.

One strategy launched a grid of $M$ blocks (one per cluster). Individual threads were responsible for the different elements in the matrices, rather than all of the threads’ working together on a
__global__ void
mstep_covariance(float* data, clusters_t* clusters, int D, int M, int N)
{
    int m = blockIdx.x;
    int row, col = compute_row_col(blockIdx.y);
    __shared__ float means[2*B]; // 2 for row,col dimensions
    copy_means_to_smem(clusters.means,means);
    __syncthreads();
    float sum = 0.0f;
    __shared__ float temp_sums[NUM_THREADS*B];
    for(int n=tid; n < N; n += NUM_THREADS)
        sum += (data[row*N+n]-means[row])*
                (data[col*N+n]-means[col])*
                clusters->memberships[m*N+n];
    temp_sums[tid] = sum;
    __syncthreads();
    sum = parallelSum(temp_sums);
    if(tid == 0) {
        clusters->R[m*D*D+row*D+col] = sum;
        clusters->R[m*D*D+col*D+row] = sum;
    }
}

Figure 4.18: M-step covariance pseudo-code

single element. This helps reduce the total amount of global memory accesses, since the input
data can be read into shared memory once per cluster, and then used by each thread. However
this implementation has the downside that on lower dimensional data sets, there simply are not
enough elements available per matrix, and the number of active threads in the block is too small for
good performance. In addition, if the number of elements is not an even multiple of the number of
threads then many of the threads sit idle and performance is degraded. Although this implementation
exceeded the performance of Figure 4.18 by roughly 20 percent for specific data sets, it does not
perform reliably across a wide range of dimensionality.

Kumar et al. [3] compute covariance by first generating a $N \times M D$ matrix of all variances. They
then use CUDA BLAS algorithms to multiply the variance by the matrix of fuzzy memberships and
in the end reduce it to a set of diagonal covariance matrices. As seen already in the C-means
algorithm, the SGEMM implementation can be fast and scales very nicely as either $M$ or $D$ grows,
but it is not optimized for rectangular matrices. The memory requirements for the entire variance
matrix is also too large for big data sets. With 100 clusters, 24 dimensions, and $10^6$ events, it would
require over 9 GB of GPU memory for the variance table alone.

There is a fourth kernel launched after the three M-step kernels that inverts the covariance ma-
trices, computes $\pi$ (the cluster probabilities), and computes a constant that is shared by all of the
likelihood calculations in the E-step. It is launched with a grid of $M$ blocks — one per cluster.
For simplicity each GPU is computing all of the constants. Otherwise the clusters would be split between the GPUs, the result copied back to the host, and then all results copied to each GPU from the host. The constants kernel is not computationally intensive compared to the other kernels, and there is no need to distribute the work and thereby suffer the additional overhead.

The covariance matrices are inverted by first performing an LU factorization. It is then necessary to compute the inverse of the $L$ and $U$ matrices, each of which is trivial via back-substitution since the matrices are triangular. Finally the inverse of the original can be computed as the product of $U^{-1}$ and $L^{-1}$ due to the mathematical property that $(LU)^{-1} = U^{-1}L^{-1}$.

Computing the LU decomposition also has the added benefit that the determinant of the matrix is the product of the diagonal values of the lower triangular $L$ matrix. The determinant is required for the calculation of the shared constant used by the E-step. The aforementioned constant is computed with (Equation 4.10), which is simply the first part of Equation (3.4).

\[
\frac{1}{(2\pi)^{M/2} \cdot \det(R)^{1/2}}
\]  

(4.10)

### 4.3.5 Combine Gaussians

The algorithm attempts to find the ideal number of clusters using a form of top-down agglomerative hierarchical clustering. After the EM stage is completed, the two most similar Gaussians in the mixture model are merged together. This is done entirely by the root node’s master thread. It involves only some simple weighted combinations of Gaussian model parameters and memory copying and thus was not parallelized. Section 3.1.2 describes the process and equations in detail. Once the new set of clusters has been determined, the root broadcasts the model parameters to all nodes. Each thread then copies the model parameters from shared memory on the host to its GPU and begins another round of EM.

### 4.3.6 Summary

This chapter presented a model for distributing data clustering tasks to parallel resources ranging from a single node with a single graphics co-processor to many nodes with multiple GPUs per node in a high performance computing environment. A hybrid of CUDA, OpenMP, and MPI provides an efficient distribution of work to the GPU co-processor, multiple GPUs on the same node, and nodes in a cluster. Both the C-means and EM with Gaussian algorithms have been optimized for execution on NVIDIA GPUs with the Tesla architecture while exhibiting sufficient parallelism that should map well to newer generation architectures with more cores and higher memory bandwidth. Notable improvements were made to the single GPU implementations compared to previous work in the field, and using coordination between multiple GPUs is novel for these algorithms. The next chapter analyzes both algorithms based on a variety of different metrics including device resource
consumption, single node performance with a wide range of input parameters, and scalability with multiple nodes using both fixed problem size and time-constrained scaling models.
Chapter 5

Results

Extensive performance testing has been performed and iterative improvements were made to the algorithm designs before arriving at the final implementations discussed in the previous chapter. 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 [39]) results are analyzed.

Correct functionality was verified using synthetically generated data (so that the answer is known). After verifying that the algorithms could correctly identify synthetically generated data, real flow cytometry data sets were analyzed.

5.1 Testing Environment

Three testing environments were used for the work in this thesis. 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 [40] 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 [40].

The final testing environment was used for testing the MATLAB versions of the two algorithms. It was a desktop machine with an Intel Q9600 quad-core operating at 2.67 GHz with 4 GB of DDR2 memory running 64-bit Windows 7 and 64-bit MATLAB R2009a.

There were three separate implementations for each of the two clustering algorithms. The first implementation operates with a single GPU. It does not contain any overheads associated with combining results from multiple threads and GPUs. The second implementation adds OpenMP and uses all GPUs available on a single node. Both of these implementations were tested on the RIT sever and a single node on the NCSA Lincoln server. The third implementation is a hybrid of MPI and OpenMP and tested exclusively on the Lincoln server with multiple nodes; however the implementation is not specific to that server and should work on any cluster environment supporting both MPI2 and OpenMP.

## 5.2 Resource Utilization

One of the strengths of NVIDIA’s Tesla architecture is its ability to handle massively parallel kernels efficiently with thousands of blocks and light-weight threads. Unlike a conventional operating system, which swaps tasks or threads by saving and loading processor state, no context switching overhead is required on the GPU. Instead, all active blocks (and their threads) coexist simultaneously within the device resources. This limits the overhead to scheduling which thread warp runs.

One common metric for CUDA kernels is occupancy. It is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. The number of active warps is a function of register usage, shared memory usage, the number of threads in each block, and the device’s compute capability. Maximizing occupancy helps hide the latency of global memory and read-after-write register dependencies since other warps can be executed while another is paused or stalled [41]. NVIDIA provides an occupancy calculator to help developers choose the proper number of threads per block. It also indicates the bottlenecks (registers or shared memory) that
should be addressed in order to achieve higher occupancy.

The following tables contain the resource details for the kernels in each of the algorithms. The first table for each algorithm reports the number of blocks for each kernel launch, the number of threads per block, the number of registers required by each thread, the shared memory consumed per block, and occupancy on a 1.3 compute capability device (like the C1060 and GTX 260). Block count and shared memory is presented as a function of the input parameters. The second set of tables shows the resource consumption for a typical flow cytometry file with 24 dimensions and 100 clusters.

Table 5.1 contains the kernels for the C-means implementation. Register usage and shared memory usage for all three kernels is relatively low resulting in high occupancy. For the first two kernels both 256 threads per block and 512 threads per block provide full occupancy (1024 active threads and 32 warps). However, 512 threads provided slightly better performance. The 512 thread case has half as many blocks and requires loading the centers into shared memory half as many times. The Centers kernel is unrolled by a factor of four, and thus its shared memory usage is proportional to the number of threads multiplied by 16 (four 32-bit values per thread). 256 threads keeps the shared memory usage low enough for 3 blocks to be active, while providing plenty of threads for each block.

Shared memory usage for the C-means kernels does not depend heavily on the size of the input. Only the distance kernel’s shared memory changes relative to the input size. The maximum number of dimensions that can be supported with the current implementation is about 4,000 — it supports a very large range of inputs. The device’s global memory becomes the bottleneck for supporting very large files, not the kernel resources. Table 5.2 shows the number of blocks and shared memory usage with a typical flow cytometry file. The shared memory consumption is slightly higher than the formulas in Table 5.1 since CUDA uses some shared memory for function arguments and constants.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Blocks</th>
<th>Threads/Block</th>
<th>Registers/Thread</th>
<th>SMEM/Block</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>$M \times \lceil N/512 \rceil$</td>
<td>512</td>
<td>6</td>
<td>$D \times 4$</td>
<td>1.00</td>
</tr>
<tr>
<td>Membership</td>
<td>$\lceil N/512 \rceil$</td>
<td>512</td>
<td>7</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>Centers</td>
<td>$D \times \lceil M/4 \rceil$</td>
<td>256</td>
<td>12</td>
<td>4096</td>
<td>0.75</td>
</tr>
<tr>
<td>Sizes</td>
<td>$M$</td>
<td>512</td>
<td>6</td>
<td>2048</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 5.3 shows resource utilization for the expectation maximization implementation. The EM kernels have slightly higher register usage than C-means, but still allow for 100% occupancy on 1.3 devices with most kernels. For 1.0 and 1.1 devices the occupancy falls to 67% for the kernels with more than 10 registers and 33% for the covariance kernel. This is still a high occupancy value and does not adversely affect performance. NVIDIA’s best practices guide [41] claims that anything above 50% occupancy will not lead to significantly greater performance, and testing in thesis agrees
<table>
<thead>
<tr>
<th>Kernel</th>
<th>Blocks</th>
<th>SMEM/Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>102400</td>
<td>136</td>
</tr>
<tr>
<td>Membership</td>
<td>1024</td>
<td>32</td>
</tr>
<tr>
<td>Centers</td>
<td>600</td>
<td>4144</td>
</tr>
<tr>
<td>Sizes</td>
<td>100</td>
<td>2080</td>
</tr>
</tbody>
</table>

Table 5.2: C-means Kernel FCS File Resource Usage

with that claim. The covariance kernel has a very large register consumption since its inner loop is
unrolled by a factor of 6 and each statement requires 3 address calculations. Testing was performed
with a variety of different unrolling factors and six was the best. The constants kernel is not a
computational bottleneck and was not parallelized. It uses only a single warp and therefore has
poor occupancy, but this is not due to resource constraints unless $D$ becomes large.

Table 5.3: EM Kernel Resource Usage

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Blocks</th>
<th>Threads</th>
<th>Register</th>
<th>SMEM</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>1</td>
<td>256</td>
<td>15</td>
<td>$D \times 8$</td>
<td>1.0</td>
</tr>
<tr>
<td>estep1</td>
<td>$16 \times M$</td>
<td>512</td>
<td>16</td>
<td>$D^2 + D \times 4$</td>
<td>1.0</td>
</tr>
<tr>
<td>estep2</td>
<td>16</td>
<td>512</td>
<td>14</td>
<td>2048</td>
<td>1.0</td>
</tr>
<tr>
<td>mstep_n</td>
<td>$M$</td>
<td>256</td>
<td>8</td>
<td>1024</td>
<td>1.0</td>
</tr>
<tr>
<td>mstep_mean</td>
<td>$M \times D$</td>
<td>256</td>
<td>10</td>
<td>1024</td>
<td>1.0</td>
</tr>
<tr>
<td>mstep_covar</td>
<td>$[M/6] \times D(D + 1)/2$</td>
<td>256</td>
<td>29</td>
<td>6192</td>
<td>0.5</td>
</tr>
<tr>
<td>constants</td>
<td>$M$</td>
<td>32</td>
<td>14</td>
<td>$D^2 \times 4$</td>
<td>0.18</td>
</tr>
</tbody>
</table>

The SMEM usage of estep1 and constants is affected significantly by dimensionality since they
store $D \times D$ covariance matrices in shared memory. The implementation can support up to only 63
dimensions before exceeding the 16KB of shared memory on the device. Shared memory usage for
typical and maximum dimensions are in Table 5.2.

Table 5.4: EM Kernel Shared Memory Usage

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Blocks</th>
<th>SMEM</th>
<th>Blocks</th>
<th>SMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>1</td>
<td>244</td>
<td>1</td>
<td>540</td>
</tr>
<tr>
<td>estep1</td>
<td>3200</td>
<td>2448</td>
<td>3200</td>
<td>16176</td>
</tr>
<tr>
<td>estep2</td>
<td>16</td>
<td>2104</td>
<td>100</td>
<td>2104</td>
</tr>
<tr>
<td>mstep_n</td>
<td>100</td>
<td>1068</td>
<td>100</td>
<td>1068</td>
</tr>
<tr>
<td>mstep_mean</td>
<td>2400</td>
<td>1068</td>
<td>6300</td>
<td>1068</td>
</tr>
<tr>
<td>mstep_covar</td>
<td>5100</td>
<td>6236</td>
<td>34272</td>
<td>6236</td>
</tr>
<tr>
<td>constants</td>
<td>100</td>
<td>2344</td>
<td>100</td>
<td>15916</td>
</tr>
</tbody>
</table>
Once shared memory usage exceeds 8192 bytes (which occurs at 45 dimensions), the number of active blocks per multiprocessor is only one, and occupancy falls to 25% if the threads per block is 256. With 512 threads per block 50% occupancy can still be achieved. An occupancy of only 25% hurt the performance of the e-step kernel significantly; however with 512 threads and 50% occupancy the performance is nearly identical to the lower dimensional case. Performance with 50% occupancy was about 10% slower than 100% occupancy. M-step kernels are not affected by the number of dimensions and seemed to perform the same or slightly better with 256 threads rather than 512. Table 5.5 summarizes experimental results for determining the optimal number of threads. 512 threads are all-around better for the E-step. 256 threads are better for the M-step on the C870, and negligibly slower on the GTX260.

Table 5.5: EM Kernel Performance with Different Threads per Block

<table>
<thead>
<tr>
<th>Device</th>
<th>C870 (1.0)</th>
<th>GTX260 (1.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensions</td>
<td>24</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>63</td>
</tr>
<tr>
<td>Threads</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>E-step (ms)</td>
<td>98</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>112</td>
<td>138</td>
</tr>
<tr>
<td>M-step (ms)</td>
<td>112</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>62</td>
</tr>
</tbody>
</table>

5.3 Single Node Performance

This section discusses performance results for the C-means and Expectation Maximization using Gaussian mixture model algorithms. Profiling information was collected using CUDA event objects from the runtime API. A custom timer library was created that mimics the timer functionality of the cutils package in the CUDA SDK but in a thread-safe manner so that it can be used with OpenMP. For each algorithm, separate timers were used for I/O (reading input files and writing output files), CPU processing (memory management on the host, combining results from multiple GPUs, OpenMP thread synchronization, and computations not performed on the GPU), memory copying to and from the GPU, GPU kernel execution, and MPI operations.

5.3.1 Algorithm Complexity

A suite of tests was designed to assess how the actual implementation performance scales as the input file and clustering parameters change compared to the computational complexity of the algorithms. C-means is $O(NMD)$ and Expectation Maximization with Gaussians is $O(NMD^2)$. These parameters are:

1. $N$ — the number of events (data points) in the input file,

2. $D$ — the number of dimensions in each event,
3. $M$ — the number of clusters.

Each variable was swept through a wide range of values while the other two variables were held constant. An input file was generated using MATLAB containing a few synthetic 64-dimensional Gaussians with a total of 4 million data points. When varying the number of events or the number of dimensions the testing scripts generate a truncated intermediate file that has the proper dimensions.

The number of iterations is held constant, and the time per iteration does not depend upon the content of the randomly generated data for these algorithms. Therefore the per iteration performance of synthetic data is identical to real-world data; the primary difference in total execution time is the number of iterations required to converge, which varies with different data sets. Time per iteration is measured by taking the average of 100 kernel executions. Using synthetic data makes it possible to easily verify the correctness of the implementations since they were generated based on known Gaussian parameters.

**Number of Events - $N$**

Both the C-means and EM algorithms have a computational complexity that is linear with respect to $N$. The number of events was varied from 250 up to 4 million while the dimensionality was held constant at 24 and the number of clusters was 100. The EM Gaussian algorithm can iteratively combine clusters and attempt to find the solution with the best likelihood, but for this testing the number of clusters was fixed and none were merged. The GTX260 did not have enough device memory for 2 million data points; similarly the C870 could not handle 4 million. Device memory consumption is equal to approximately $4N(M + D)$ bytes; refer to Chapter 4 for exact details.

Figure 5.1 shows the total execution time, and Figure 5.2 shows the time per iteration as the number of data points increases for a variety of different GPU configurations. Both the total time and the time for each iteration increase linearly when the number of data points is above about 8000. For smaller values of $N$ there is not enough data for all of the multiprocessors and threads to be in use. Therefore the GPU iteration time increases sub-linearly for small values of $N$.

The C870, GTX260, and C1060 data series use the single GPU implementation while the C1060x2 uses the OpenMP enhanced version and two C1060 GPUs. The total execution time figure includes all I/O, data structure creation on the host, memory copying, and GPU kernel time. In the C1060x2 case it also includes all thread creation, synchronization, and reduction of results from both GPUs.

The C870 is the slowest of the four GPU configurations — it has the fewest number of cores and the lowest memory bandwidth. The C1060 has more cores than the GTX260 (240 vs. 192) and operates at a higher clock rate; however it actually performs about 15% slower than the GTX260. This is because the GTX260 has higher memory bandwidth — about 10% higher peak theoretical bandwidth, but measures about 25% faster with the NVIDIA SDK bandwidthTest application. Both
Figure 5.1: C-means total time scalability: N

Figure 5.2: C-means GPU kernel scalability: N
the C-means and EM GPU implementations are memory bound since not much computation is performed per memory access.

Adding an additional C1060 GPU splits the number of events per GPU in half and doubles the effective number of cores and memory bandwidth. Therefore the C1060x2 has about half the runtime of the C1060. The time per iteration (Figure 5.2) has a nearly ideal speedup with 2 GPUs vs 1 GPU — the speedup fluctuates between 1.97 and 2.02 once the number of events is 64000 or greater. Since the entire algorithm is not parallelized (e.g. I/O, host memory management, and host computation) and there are some small overheads to combine results from multiple GPUs, the overall speedup for two GPUs is about 1.90 for the test in Figure 5.1.

![EM Iteration time on various GPUs](image)

**Figure 5.3: Expectation Maximization Scalability: N**

Figure 5.3 shows the effect of increasing \( N \) for the EM algorithm. The time per iteration has more fluctuation than C-means as \( N \) increases, but it is still linear. Partition camping is the most likely cause for the oscillations in the execution time. Different combinations of \( B \) and \( N \) change the distribution of active warps to the different memory partitions. Increasing \( B \) improves the linearity slightly on some GPUs, but on average decreases performance particularly with smaller values of \( N \).

A comparison of the time spent on different portions of the program execution can be found in Figure 5.4. 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 is a larger portion of the overall execution time. Note that the vertical axis is log-scale so that the overhead is more visible. The GPU kernel time is based on 100 iterations. 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.

![Graph showing C-means and EM execution breakdown](image)

Figure 5.4: Breakdown of Execution Time for C-means and EM: N

**Number of Clusters - M**

A naive parallel implementation of C-means has a computational complexity proportional to $M^2$. However, this can be optimized to $M$ as discussed in Section 4.2. EM also has a computational complexity that grows linearly with respect to $M$. Testing was performed with the number of data points held constant at 500K with 24 dimensions while the number of clusters is varied from 1 to 256.

Figure 5.5 shows the time per iteration using the naive quadratic parallel implementation of C-means. The optimized implementation in Figure 5.6 is linear and has a speedup over the standard implementation for this dataset of about 6 for 250 clusters. The speedup of the linear version is not directly proportional to $M$ since only the membership kernel is affected — the distance and centers kernels were already linear regarding $M$. The EM implementation also has a smooth linear trend as $M$ is increased as seen in Figure 5.7.

As seen above with the number of events, Figure 5.8 shows what fraction of the total execution
Figure 5.5: Naive C-means Scalability: M

Figure 5.6: Optimized C-means Scalability: M
Figure 5.7: Expectation Maximization Scalability: M

Figure 5.8: Breakdown of execution time for C-means and EM: M
time is spent on the different portions of the algorithms as the number of clusters increases. It is noteworthy that as $N$ or $M$ increased, the overhead remained a relatively steady percentage of the overall execution time. In other words, the overhead grew proportional to the input size or the number of clusters. With small numbers of clusters, (below about 10), the overhead is a large percentage of the total execution time. This overhead suggests that the problem will not scale well to multiple nodes if the number of events or clusters is very small. Similarly, since the overhead does not continue decreasing for very large numbers of events or clusters, increasing the problem size will not continue to increase scalability (unless the larger data sets require more kernel iterations). From the figure it is also clear that C-means (about 6% overhead) will not scale as well as EM (0.5% overhead). This will be discussed in more detail in the section on horizontal scalability.

**Number of Dimensions - $D$**

The computational complexity of C-means with a Euclidean distance measure grows linearly as the number of dimensions increases. EM with Gaussians computes likelihood, (its similarity measure), as a multiplication of a $D$ dimensional vector with a $D \times D$ covariance matrix; thus the time-complexity is proportional to $D^2$. Testing was performed using a 63-dimensional data set with 500K events and 100 clusters. The number of dimensions was varied from 1 to 63 while the other parameters were constant. C-means supports a much higher number of dimensions, but the EM implementation is limited to 63 dimensions due to shared memory usage.

Figure 5.9 shows that the C-means implementation scales linearly with dimension, but there are some slight irregularities. Memory accesses are coalesced regardless of the dimension of the input data. Partition camping and the number of blocks, (which varies with dimension), not being an even multiple of the number of multiprocessors account for the small fluctuations in the linear trend. The relative performance of the four GPU configurations is comparable to the $N$ and $M$ tests; the C870 is the slowest, and the GTX260 is faster than the C1060 because of its higher memory bandwidth. Finally the use of two C1060 GPUs is the fastest and has an overall speedup compared to a single C1060 including all overheads ranging from 1.83 for one dimension up to 1.95. The speedup of the GPU kernel ranges from 1.94 to 1.99.

EM has a quadratic increase in execution time when the number of dimensions increases as seen in Figure 5.10. This accounts for the considerably higher time per iteration for EM clustering compared to C-means with data that is not low-dimensional. The design of the parallel implementations for both algorithms exposes sufficient parallel tasks to ensure that performance scales well across a wide range of dimensionality. This design makes the implementations suitable for a wide range of applications — not just flow cytometry data.

Figure 5.11 shows the breakdown of execution time for both C-means and EM with Gaussians as $D$ increases. Unlike $N$ and $M$, overhead continues to decrease as a percentage of total execution
Figure 5.9: C-means Scalability: D

Figure 5.10: Expectation Maximization Scalability: D
time as the dimensionality increases. This makes sense since the size of the data (and the overhead for I/O, allocating data structures, and memory copying) grows linearly, but the computational complexity grows quadratically.

![C-means C1060x2 Execution Breakdown](image1)

![EM C1060x2 Execution Breakdown](image2)

Figure 5.11: Breakdown of execution time for C-means and EM: D

### 5.3.2 Speedup

The common trend in GPGPU research and commercial applications is to compare the performance of the GPU accelerated program to the same algorithm running on just a CPU. The previous section has already displayed performance figures for various data sizes and clustering parameters. This section compares those same numbers to various CPU versions.

The first is a reference version written in C. The computations are reorganized in order to avoid any redundancy that may be required for parallelization on the GPU version. Data structures and data access patterns are organized to maximize spatial locality; therefore benefiting from the CPU’s cache. This provides a fairer representation of the algorithm on a CPU, but it is not heavily optimized. For example, SSE instructions and libraries such as Intel MKL were not explored. The algorithms are also only single threaded and therefore only make use of one CPU core.

A second comparison is performed between each algorithm and its corresponding Mathworks MATLAB implementation on the CPU. The `fcm` function in the Fuzzy Logic Toolbox performs C-means clustering and the `gmdistribution.fit` function from the Statistics Toolbox does Expectation Maximization with Gaussians.
Figure 5.12: C-means Kernel Speedup vs. events: C reference version

Figure 5.13: C-means Speedup vs. events: C reference version
C-means

Figure 5.12 shows the speedup of the computational portion (the GPU kernels) compared to the C-means CPU kernels with 24 dimensions and 100 clusters. The best speedup of the three single-GPU configurations is the GTX260 with a maximum of 110x. The speedup is 106x on average when the number of data points is 16,000 or greater. The C1060x2 configuration, (a single node of the NCSA Lincoln server), has an average kernel speedup of 178x for 16,000 or more events.

The computational speedup, while interesting, is not always an accurate representation of the speedup of the entire GPU-accelerated algorithm. There are two major factors that degrade the overall speedup of a GPGPU algorithm compared to a CPU-only counterpart. First, there are generally some computations that cannot be accelerated on the GPU, such as I/O. Second, there is additional overhead required to copy code, data, and results to and from the GPU. The overall speedup is computed by dividing the total execution time of the CPU program by the total time of the GPU program. The results are displayed in Figure 5.13. Maximum speedup for the GTX260 is 104x, with an average of 90x for \( N \geq 16,000 \). The C1060x2 has an average of 152x for \( N \geq 16,000 \). For large numbers of events the overall speedup ranges from 5% less than the kernel speedup with the C870, to 14% less with the C1060x2. This reduction is comparable to the overhead seen in Figure 5.4. The GPUs with the fastest kernels have the largest percentage drop in overall speedup (compared to their corresponding computational speedup) because the GPU computation is a smaller percentage of the overall execution time given a fixed problem size.

Many of MATLAB’s functions are heavily optimized with libraries written in C/Assembly and use LAPACK for linear algebra operations. However, the MATLAB FCM function does not appear to be very efficient. It performs significantly slower than the C reference version used in the comparisons above. Nonetheless, MATLAB is a widely used tool. Figure 5.14 shows the overall speedup of the GPU version versus the MATLAB `fcm` function. Maximum speedup is over 300x for the GTX260 and over 500x with two C1060s. The FCM function does not appear to use any MATLAB parallelization or vectorization — there is no difference in performance between the processor affinity being set to one core or four cores.

The memory requirement of the MATLAB version is very large. With 500K events MATLAB was using over 2GB of memory. With 1 million events it was using 3.5GB of physical memory and swapping from disk (the MATLAB testing system only had 4GB of memory) — results with memory swapping were excluded from Figure 5.14.

Figures 5.15 and 5.16 show how speedup is affected by dimension. The GPU speedup decreases as the number of dimensions increases. At first it appears strange that the GPU performs poorer relative to the CPU reference and MATLAB when the amount of work and potential parallelism increases. Unlike small values of \( N \), which leave GPU resources underutilized, with 100 clusters and 500K events there are plenty of parallel blocks to fill the resources of the GPU, even with only one dimension. The CPU version benefits significantly from spatial locality and caching when
Figure 5.14: C-means Speedup vs. events: MATLAB fcm

Figure 5.15: C-means Speedup vs. dimension: C reference
dimensionality increases (by reducing accesses to system memory), whereas the GPU has a linear increase in the number of global memory accesses required as $D$ increases. The CPU cache has a limited size, so as dimensionality continues to increase the GPU speedup levels off.

Figures 5.17 and 5.18 show how speedup is affected by the number of clusters. Speedup increases with more clusters, but quickly levels off after about 10 clusters. Unlike the input data, which is used repeatedly and benefits from caching, the membership values are used only once. Therefore the GPU is not at a disadvantage and speedup does not decrease when the number of clusters increases.

**Expectation Maximization**

Figures 5.19 and 5.20 show the computational speedup and overall speedup of the EM program compared to the C reference version running on a CPU. Unlike C-means, the overall speedup is very similar to the computational speedup. As seen in Figure 5.4, the GPU kernels for EM encompass a larger percentage of the execution time — over 99% instead of about 86%. The 1% overhead does not impact overall speedup nearly as much as 14% overhead.

The best speedup out of the three single GPU configurations is the GTX260 ranging from 58x to 84x and an average of 73.5x for $N \geq 16,000$. Even with data sets as small as 1,000 the GPU has a speedup over the CPU reference of 10x. With two C1060 GPUs (a single Lincoln node) the
Figure 5.17: C-means Speedup vs. clusters: C reference

Figure 5.18: C-means Speedup vs. clusters: MATLAB
Figure 5.19: EM Kernel Speedup vs. events: C reference version

Figure 5.20: EM Speedup vs. events: C reference version
speedup ranges from 93x to 145x with an average of 120x for $N \geq 16,000$. Neither the GPU nor CPU implementations have perfectly linear performance with respect to $N$. The general trend is that larger data sets have higher speedup up until about 64,000 data points where speedup levels off, but it is not monotonically increasing due to the fluctuations in performance with certain values of $N$.

Despite the higher computational complexity of the EM algorithm compared to C-means, the GPU implementation has a smaller speedup compared to its CPU reference version. The EM program exhibits a large amount of parallelism, but computing simple statistics such as means and covariances over large data sets makes the algorithm memory-bound. The GPU implementation exploits data independence to compute likelihood values and statistics for various clusters and dimensions in parallel, but this comes at the cost of repeatedly accessing the same input data without the benefit of caching. The EM algorithm has a lot of spatial locality since mean vectors and covariance matrices are accessed repeatedly, which the CPU’s more sophisticated memory hierarchy can exploit via caching. The GPU implementation uses shared memory to cache covariance matrices in high-speed memory, but shared memory is a very limited resource so only a single covariance matrix can be stored in each kernel block. Future generations of GPU architectures, such as NVIDIA’s upcoming Fermi [28], which provide caching of the device’s global memory will hopefully give the GPU an even greater advantage over CPUs with memory-bound algorithms.

MATLAB provides a Gaussian mixture model fitting function using Expectation Maximization in the Statistics Toolbox. In contrast to the `fcm` function, the MATLAB EM algorithm appears to be very optimized. Figure 5.21 shows the speedup of the GPU implementation compared to MATLAB’s `gmdistribution.fit` function. Speedup versus the MATLAB version is about 60% of the C reference version. The average speedup is 44x with the GTX260 and 73x with two C1060s for $N \geq 16,000$. Like the `fcm` function, the MATLAB EM implementation uses significantly more memory than the GPU and C reference implementations and could not handle data sizes larger than about 500,000 on a system with 4GB of memory.

Figures 5.22 and 5.23 show how speedup is affected by the number of dimensions. Like with C-means, speedup is inversely proportional to dimension. For the first 10 to 15 dimensions the decrease is rather dramatic. Speedup then begins to level off and decrease gradually. For one dimensional data the speedup for the EM C reference version is 724x for the GTX260 and 1090x for two C1060s. These speedup values are high for a number of reasons. First, computational density for the E-step is reasonably high because it involves various exponentials and logarithms (which are much more time consuming than simple multiplication and addition operations). These operations which can be done in parallel on the GPU and overlapped with memory accesses from other threads. Second, the membership portion of the E-step, which converts likelihood values into fuzzy membership values, involves a lot of non-spatial memory access (the memory layout is optimized for the likelihood kernel). This makes the performance of that kernel quite poor on
Figure 5.21: EM Speedup vs. events: MATLAB Gmdistribution.fit EM

Figure 5.22: EM Speedup vs. dimension: C reference
the CPU. The membership function has a complexity of $O(NM)$. Therefore, as $D$ increases its runtime becomes less significant compared to the more optimized $O(NMD^2)$ functions in the algorithm. Furthermore, the number of times that the input data and the membership matrix needs to be traversed grows as $D$ increases. The CPU has the benefit of caching to help reduce access to system memory; the GPU does not have caching and must rely on coalescing and a large memory bandwidth. Once bandwidth is maximized on the GPU, processing cores idle waiting for data to be loaded from memory and instruction throughput decreases.

Finally, Figures 5.24 and 5.25 show the speedup as the number of clusters increases. Speedup increases with more clusters, but plateaus after 10 to 20 clusters. Speedup fluctuates a lot with a small number of clusters for a couple reasons. First, some kernels do not have enough blocks for full device occupancy and the number of blocks is not an even multiple of the number of multiprocessors. Secondly, the covariance kernel is loop-unrolled by a factor of six; therefore performance is not optimal if the number of clusters is not divisible by six. This becomes less significant for a large number of clusters because it only affects the blocks for the final group of clusters.
Figure 5.24: EM Speedup vs. clusters: C reference

Figure 5.25: EM Speedup vs. clusters: MATLAB
5.4 Comparison to prior work

The Background chapter of this thesis surveyed the previous work on GPU-based acceleration of the C-means and EM with Gaussian algorithms. Chapter 4 discussed some of the improvements this thesis made compared to the existing implementations. This section compares the performance of the current implementations in this thesis to the previous work. Unfortunately exact comparisons cannot be made due to differences in available hardware. 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.

5.4.1 C-means

In 2009 Espenshade and Pangborn et al. published a paper on a single-GPU implementation of fuzzy C-means [1]. Speedups of 84x were observed, but the CPU version naively replicated many of the flaws in the GPU version. The algorithm improvements made in this thesis were already discussed in Chapter 4.

Table 5.6 shows a comparison of the current C-means implementation to the old implementation in [1] with identical problem sizes and the same testing hardware. The data size is 100K events with 16 dimensions. The new GPU version in this thesis has an improvement over the previous GPU version \((t_2/t_3)\) 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 [1]. The new implementation also coalesces more global memory access and has higher kernel occupancy.

<table>
<thead>
<tr>
<th>(M)</th>
<th>CPU ((t_1))</th>
<th>Espenshade et al. ((t_2))</th>
<th>GPU ((t_3))</th>
<th>Speedup ((t_1/t_2))</th>
<th>Speedup ((t_1/t_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>92.4</td>
<td>11.88</td>
<td>1.27</td>
<td>7.8</td>
<td>73.1</td>
</tr>
<tr>
<td>8</td>
<td>183.15</td>
<td>15.35</td>
<td>1.74</td>
<td>11.9</td>
<td>107.2</td>
</tr>
<tr>
<td>12</td>
<td>267.62</td>
<td>21.61</td>
<td>2.05</td>
<td>12.4</td>
<td>130.6</td>
</tr>
<tr>
<td>16</td>
<td>339.89</td>
<td>26.36</td>
<td>2.56</td>
<td>9.3</td>
<td>133.2</td>
</tr>
<tr>
<td>24</td>
<td>540.03</td>
<td>51.40</td>
<td>4.07</td>
<td>10.5</td>
<td>132.8</td>
</tr>
<tr>
<td>32</td>
<td>728.08</td>
<td>60.69</td>
<td>5.67</td>
<td>12.0</td>
<td>125.3</td>
</tr>
<tr>
<td>48</td>
<td>1071.73</td>
<td>116.09</td>
<td>7.69</td>
<td>9.2</td>
<td>139.7</td>
</tr>
<tr>
<td>60</td>
<td>1353.69</td>
<td>204.36</td>
<td>9.72</td>
<td>6.6</td>
<td>138.7</td>
</tr>
</tbody>
</table>
The next C-means paper is by Shalom et al. [35]. 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 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. However, convergence in this thesis is somewhat stricter since it ensures that the total change in cluster position is less than the threshold, whereas Shalom et al. stops when the maximum movement of a cluster center is less than epsilon. 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 [34] [2]. The paper indicates that the testing was performed with 100 iterations on synthetic data with random cluster centers. Table 5.7 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 the C-means 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>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>D</th>
<th>Anderson et al. ($t_1$)</th>
<th>C-means ($t_2$)</th>
<th>Speedup ($t_1/t_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4096</td>
<td>4</td>
<td>39</td>
<td>12</td>
<td>3.25</td>
</tr>
<tr>
<td>4</td>
<td>4096</td>
<td>128</td>
<td>53</td>
<td>51</td>
<td>1.04</td>
</tr>
<tr>
<td>64</td>
<td>4096</td>
<td>4</td>
<td>172</td>
<td>53</td>
<td>3.24</td>
</tr>
<tr>
<td>64</td>
<td>8192</td>
<td>4</td>
<td>362</td>
<td>78</td>
<td>4.64</td>
</tr>
<tr>
<td>16</td>
<td>40960</td>
<td>32</td>
<td>471</td>
<td>331</td>
<td>1.42</td>
</tr>
<tr>
<td>4</td>
<td>409600</td>
<td>8</td>
<td>780</td>
<td>529</td>
<td>1.47</td>
</tr>
</tbody>
</table>
5.4.2 EM with Gaussians

Kumar et al. published a paper on Expectation Maximization with Gaussians in June 2009 [3]. Their implementation supports only diagonal covariance, so the version in this thesis 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 nearly identical to the Tesla C1060 except that it is sold for workstations and has display output ports.

Table 5.8 shows the time per iteration from the Kumar et al. paper compared to the diagonal-only version of EM in this thesis. 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. The SGEMM function also performs poorly on highly rectangular matrices as seen above in Figure 4.11.

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 [4]. 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 thesis. 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 thesis 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 5.9 compares the CPU time for the Harp CPU reference version to the version of EM in this thesis 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 thesis 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.

Table 5.9: Comparison of EM time (s) to [4]

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>D</th>
<th>Harp CPU Reference ($t_1$)</th>
<th>GTX260 ($t_2$)</th>
<th>Speedup ($t_2/t_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>10</td>
<td>8</td>
<td>2.5</td>
<td>0.28</td>
<td>8.8</td>
</tr>
<tr>
<td>10K</td>
<td>10</td>
<td>8</td>
<td>14.5</td>
<td>0.29</td>
<td>49.5</td>
</tr>
<tr>
<td>100K</td>
<td>10</td>
<td>8</td>
<td>267</td>
<td>0.83</td>
<td>321.2</td>
</tr>
<tr>
<td>1000K</td>
<td>10</td>
<td>8</td>
<td>2772</td>
<td>6.20</td>
<td>446.7</td>
</tr>
</tbody>
</table>

5.5 Horizontal Scalability

Horizontal scalability assesses how well a parallel algorithm scales when computing resources are increased by adding more processing nodes, data servers, etc.. This is opposed to vertical scalability which adds computing resources by replacing the existing processors or memory with higher performance/capacity components. The implementations in this thesis use horizontal scaling by adding more server nodes and GPUs. Ideally adding more nodes to a cluster means the performance of the algorithm increases linearly, but this is nearly impossible in reality. Most non-trivial parallel algorithms have overhead from communication, synchronization, and data dependencies. It is also often not possible to parallelize the entire algorithm. Therefore adding additional computing resources is only effective up to a certain point since the non-parallel sections of code become a bottleneck.

Amdahl’s law is a formulation of an observation made by Gene Amdahl in 1967 [42]. The law (Equation 5.1) computes the overall speedup of an algorithm based on the fraction of it that is improved, $P$, and the speedup, $S$. This is often applied to parallel computing by replacing the fraction that can be improved by the fraction that can be parallelized, $P$, and replacing the speedup $S$ by the number of processors, $N$. Amdahl’s law can be thought of as a formulation of the law of diminishing returns for parallel computing.

$$ Speedup = \frac{1}{1 - P + \frac{P}{S}} \quad Parallel\ Speedup = \frac{1}{1 - P + \frac{P}{N}} $$

(5.1)

Taking the mathematical limit as the number of processors approaches infinity, Amdahl’s law provides an upper-bound on the speedup of an algorithm based on the percent that can be parallelized. In other words, if 90% of an algorithm is parallelized, then the maximum speedup possible even with an infinite number of processors is 1/0.1, or 10. Thus we could say that it is not cost effective to horizontally scale said algorithm by more than 10-20 processors. Figure 5.26 shows the speedup compared to ideal horizontal scaling for various values of $P$. 
Max speedup = \frac{1}{1 - P}

(5.2)

Figure 5.26: Amdahl’s law with various degrees of parallelization

Figures 5.4, 5.8, and 5.11 in the previous section showed a breakdown of the execution time for the parallel clustering implementations. Some components of the execution time are parallelized, but others are not and thus become a bottleneck when horizontally scaling to multiple nodes. I/O is not parallelized. Some of the CPU time is not parallelized and there is additional CPU overhead on the root node that grows as the number of nodes increases. The Memcpy time is reduced with multiple GPUs since each GPU handles a fraction of the original dataset; however there are some additional memory copies required to aggregate the results between the GPUs. The MPI implementation introduces additional overhead for distributing the data set and combining results from multiple nodes over a network connection. The serial portion of the total runtime and the additional overhead of the parallel implementation limit the scalability of the the algorithms on a fixed problem size due to Amdahl’s law.

5.5.1 Fixed Problem Size Scalability

Fixed problem size analysis assesses performance of the algorithm as more processors are added and the problem size does not change. It is often called either strong scaling or true speedup. The
combination of Amdahl’s law and any additional overhead that occurs from increasing the number of nodes limits the speedup. To test how well each algorithm strongly scales the number of MPI processes was increased from 1 to 64. Each MPI process has two threads and two GPUs for a total of 2 to 128 GPUs. The problem size is fixed to $10^6$ events, 24 dimensions, and 100 clusters. The number of GPU kernel iterations is fixed at 200. Clustering was repeated 10 times for each number of nodes in order to get an average value for the I/O and MPI time. MPI communication involves network access on Lincoln and seems to have a large variance.

**C-means**

Figure 5.27 shows the parallel speedup for the number of Lincoln processing nodes, each with two Tesla C1060 GPUs. The test shows that the kernel speedup is nearly ideal up through 16 nodes (32 GPUs) achieving 96% of the ideal speedup. With 64 nodes (128 GPUs) the kernel speedup is 85% efficient. Even though the kernel speedup does not include any overhead from MPI or OpenMP, with a fixed problem size the number of events per GPU continues to decrease. As seen in Figure 5.2, the GPU speedup compared to a CPU begins to decrease with small data sets. The GPU speedup plateaus at approximately 16,000 events, but with 128 GPUs there are only about 8,000 events per GPU.

![C-means Strong Scaling: M=100, D=24, N=10^6](image)

**Figure 5.27: C-means Strong Scaling Speedup**

Despite relatively efficient scaling of the computational kernel, the overall program performance
does not scale well above 16 nodes. The overhead consumes about 20% of the execution time with 16 nodes. At 64 nodes the overhead has increased to 50% and only half of the execution time is spent on meaningful computation. Figure 5.28 shows a breakdown of the percentage of total execution time for different portions of the algorithm.

![C-means Strong Scaling Time Breakdown](image)

Figure 5.28: C-means Strong Scaling Time Breakdown

The I/O category reads the binary data from disk (which is stored on a network-based file system on Lincoln) on the root node and then distributes it via MPI. For a single node, there is no MPI communication required so the input time is lower. For two or more nodes, the input time remains nearly constant. Although the data must be sent to more nodes, the data sent per node decreases by the same factor. The root node sends each message asynchronously so that it can continue preparing all of the messages without waiting for the other nodes. This proved to be far more efficient than having every node load the file from the network file system on Lincoln.

CPU initialization sets up data structures on the host and transposes the input data, which depends on the number of events per node. Therefore the CPU initialization time decreases with more nodes. The MPI communication and synchronization time on average grows as the number of nodes increases, but not in a consistent trend. There is also a lot of variance in total MPI time from one run to the next. The Infiniband network topology on Lincoln, the details of the MVAPICH implementation of MPI, the proximity of the nodes that happen to get allocated for the job by the queue, and network congestion from other concurrent jobs on the cluster all likely contribute to irregularity in the MPI overhead for the algorithm. Unfortunately a limitation on node hours available for this
Figure 5.29: C-means Strong Scaling Speedup

thesis does not permit exhaustive testing with a large number of repetitions which would help negate the impact of outliers and transient network conditions.

GPU memory copying time decreases as the number of nodes increases since each node handles fewer events and the memory copying is done in parallel. There appears to be a relatively constant cost incurred of about 30 ms for the first memory copy performed inside a new CUDA context, regardless of the size of the copy. This constant overhead causes memory copying time to decrease sub-linearly as the number of nodes increases, so its percentage of the total execution time grows.

\[ Efficiency = \frac{Parallel Speedup}{Ideal Speedup \equiv Number Of Nodes} \]  

The scalability of C-means with data sizes on the order of \(10^6 \times 10^3\) with 100 clusters is relatively poor. Parallel efficiency is a metric that assesses how the speedup with \(P\) nodes compares to the ideal speedup. Efficiency is only 90% with 16 GPUs (8 nodes) and drops to 43% with 128 GPUs (64 nodes). The time spent on 200 iterations of computation is only half a second with 64 nodes. File system I/O, network communication, and node synchronization become the bottleneck. Furthermore, the GPU becomes less efficient when the problem size per GPU continues to decrease. In other words, it is not worthwhile to use a large number of nodes for C-means clustering of flow cytometry files.
Increasing the problem size from one million events to four million events improves the scalability (see Figure 5.29). The parallel kernel speedup is almost ideal (97%) with 128 GPUs since the problem size per GPU is large enough for the GPU to have near maximum efficiency. The overall parallel speedup efficiency is 91% with 16 GPUs and 53% with 128 GPUs — 10% higher efficiency than the data set with one million events. Expanding to even larger data sets did not have a notable increase in efficiency. The GPU does not gain any additional efficiency with even larger data sets and the largest source of overhead (file I/O) grows as the data size increases.

Table 5.10 summarizes the parallel speedup of the GPU Kernel and the whole GPU algorithm with respect to a single node. The CPU column is the speedup of the MPI GPU program when compared to the CPU reference version running on a single processor core. The CUDA GPU enhanced algorithm running on 64 nodes with a total of 128 Tesla C1060 GPUs achieves a speedup over the CPU reference version of 5915x — 3.77 orders of magnitude improvement.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>M=100, D=24, N=1,000,000</th>
<th>M=100, D=24, N=4,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU Kernel</td>
<td>GPU</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.98</td>
<td>1.95</td>
</tr>
<tr>
<td>4</td>
<td>4.00</td>
<td>3.84</td>
</tr>
<tr>
<td>8</td>
<td>7.92</td>
<td>7.22</td>
</tr>
<tr>
<td>16</td>
<td>15.43</td>
<td>12.58</td>
</tr>
<tr>
<td>24</td>
<td>23.16</td>
<td>17.21</td>
</tr>
<tr>
<td>32</td>
<td>30.34</td>
<td>20.75</td>
</tr>
<tr>
<td>48</td>
<td>42.56</td>
<td>24.31</td>
</tr>
<tr>
<td>64</td>
<td>54.25</td>
<td>27.18</td>
</tr>
</tbody>
</table>

**Expectation Maximization**

Expectation Maximization with Gaussians has overhead that is very similar to C-means. However, the computational complexity of EM is higher and therefore the overhead is a smaller percentage of the total execution time on a single node. This gives EM better scalability. A test was conducted with D=24, M=100, and N=1,000,000. The number of nodes was varied from 1 to 64 (2 to 128 GPUs).

Figure 5.30 compares the speedup of the GPU kernels and the overall algorithm to ideal speedup. Figure 5.31 is the breakdown of the different overheads and sections of the algorithm. The overhead is a much smaller percentage of the execution time than in C-means for the same number of nodes and the corresponding parallel speedups are higher. Parallel efficiency for the kernel is 93% and the overall efficiency is 72% for 128 GPUs (64 nodes).

Table 5.11 summarizes the parallel speedup of the GPU Kernel and the whole GPU algorithm
Figure 5.30: EM Strong Scaling Speedup

Figure 5.31: EM Strong Scaling Time Breakdown
with respect to a single node. The CPU column is the speedup of the MPI GPU program when compared to the CPU reference version running on a single processor core. The CUDA GPU enhanced algorithm running on 64 nodes with a total of 128 Tesla C1060 GPUs achieves a speedup over the CPU reference version of 6286x — 3.8 orders of magnitude improvement.

Table 5.11: Expectation Maximization Speedup Summary

<table>
<thead>
<tr>
<th>Nodes</th>
<th>M=100, D=24, N=1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU Kernel</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.87</td>
</tr>
<tr>
<td>4</td>
<td>4.07</td>
</tr>
<tr>
<td>8</td>
<td>8.12</td>
</tr>
<tr>
<td>16</td>
<td>15.32</td>
</tr>
<tr>
<td>24</td>
<td>23.98</td>
</tr>
<tr>
<td>32</td>
<td>32.11</td>
</tr>
<tr>
<td>48</td>
<td>43.79</td>
</tr>
<tr>
<td>64</td>
<td>59.46</td>
</tr>
</tbody>
</table>

5.5.2 Time-constrained Scaling

Time-constrained scaling assesses scalability of an algorithm keeping the total execution time, assuming perfect speedup, constant [39]. In other words, the problem size grows proportional to the computing resources and ideally the execution time will remain the same. Other names for this analysis include weak scaling and scaled speedup. This form of analysis subverts the effect of Amdahl’s law on fixed problem sizes. The non-parallel fraction of the computation remains the same percentage of the total execution time, rather than becoming increasingly large relative to the time for the parallel section. Gustafson’s law [43] is a reformulation of Amdahl’s law that computes speedup based on the number of processors, \( P \), and a serial fraction \( \alpha \). It is important to realize that \( \alpha \) is the proportion of time spent on serial processing and overhead with \( P \) processors — it is not equivalent to the serial fraction in Amdahl’s law which is based on the execution time for a single processor.

\[
\text{Speedup} = P - \alpha(P - 1) \tag{5.4}
\]

In reality, the time spent on serial computation and parallel computation generally remains fixed, but overhead for distributing work, synchronization, satisfying data dependencies, and collecting results will increase with more processing nodes and larger problem sizes. Therefore even weak scaling generally has an upper-bound. For these two algorithm implementations there is very little fixed serial computation since the majority of the work is offloaded to the GPU co-processors. The
non-parallel portion of the execution time is therefore mostly overhead, which increases both as the number of nodes increase and the problem size increases. For example, larger data files cause I/O and distribution/collection of the data set to take more time. If the overhead grows proportional to \( P \), then the speedup obtained from Gustafson’s law is identical to the fixed problem size Amdahl’s law.

Execution time is ideally proportional to \( N/p \) for C-means. Therefore the problem size is scaled to \( N \times p \) for time-constrained analysis. Time-constrained analysis for C-means was performed by fixing the work to \( N=50000 \), \( D=24 \), \( M=100 \), and 200 kernel iterations per node. Seven jobs were executed with the number of nodes varying from \( 2^1 \) to \( 2^7 \). Input files with randomly generated data were created with the number of events ranging from 50,000 for a single node to 3,200,000 for 64 nodes. A final job was executed to obtain single node execution times for all of the corresponding data sizes. Weak scaling speedup was then computed by dividing the single node execution time by the execution time for the corresponding multi-node job with the same problem size.

![C-means time-constrained scaling: Execution Time](image)

Figure 5.32: C-means execution time with constant problem size per node

Ideally the execution time remains constant when the number of nodes is increased with weak scaling, which is equivalent to ideal linear scaled speedup. If execution time increases with more nodes while the work per node is held constant then the scaled speedup is sub-linear. Figures 5.32 and 5.33 show execution time for C-means and EM with Gaussians respectively. In both cases the kernel time remains nearly constant, while the total execution time increases. The C-means time
increases by a greater percentage and therefore has poorer scaled speedup.

Figures 5.34 and 5.35 display the speedup relative to a single Lincoln node. The scaled speedup is not much better than the strong scaling in the previous section. Increasing the problem size per node from 50,000 to 100,000 has a marginal increase in parallel efficiency. This effectively doubles the total problem size from 3.2 million to 6.4 million data points. Parallel efficiency increased from 58% to 61%, not near as large as the 13% increase from 1 million to 4 million data points observed in the previous section.

Weak scaling helps diminish the impact of serial computations which cannot be parallelized and do not depend upon the number of processors. The serial computation that limits speedup in Amdahl’s law is assumed to be independent of the number of processors. For these two algorithms the limiting factor is I/O, synchronization, and communication, which are all dependent on the number of processors. Table 5.12 contains a summary of the scaled speedup.

### 5.6 Results - Iris Data

The Iris data set is a small multivariate data set with four dimensions that measure different characteristics of iris flowers (sepal length, sepal width, petal width, petal length). It contains three classes corresponding to different species of the Iris flower with 50 samples each. The data were
Figure 5.34: C-means Scaled Speedup

Figure 5.35: EM Scaled Speedup
Table 5.12: Scaled Speedup Summary: M=100, D=24, N=50K per node

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

collected by Edgar Anderson and made popular by Fisher in 1936 [44] — since then it has been widely used in data classification and data mining publications. Figure 5.36 is a scatter matrix of the labeled data used in Fisher’s publication, which shows a 2-dimensional plot of the data for each permutation of the variables [11]. The first iris species (setosa, in red) is linearly separable, but the other two overlap.

The data were clustered using both algorithms. Table 5.13 shows the actual size, the C-means size, and the Gaussian size of each cluster (species). Both algorithms compute membership probabilities rather than discrete cluster assignments, which results in floating-point cluster sizes instead of integers. Table 5.14 shows the actual means (based on the known labeled data), the centers as estimated by C-means, and the means estimated by a Gaussian mixture model. The estimated C-means centers are quite close to the actual mean values with a mean squared error (MSE) of $5.9 \times 10^{-3}$. The Gaussian algorithm estimate is better with an MSE of $3.4 \times 10^{-5}$.

The Gaussian mixture model estimates are closer to the actual value than C-means. Gaussians take both mean and covariance into account, which allows the cluster shapes to be elliptical and rotated rather than simply spherical. Since the data clusters are non-spherical (see Figure 5.36), this allows a much closer approximation to the actual cluster values, even if they are not a perfect Gaussian distribution.

A problem with the expectation maximization algorithm is local minima, which can result in converging to a suboptimal solution depending on the random starting conditions. In an attempt to avoid this problem, the clustering is repeated 10 times with different random initial clusters and then a final solution is chosen based on the result with the highest likelihood value. This is the
Figure 5.36: Iris Data Set [11]
Table 5.14: Iris Data Cluster Means

<table>
<thead>
<tr>
<th>Iris Species</th>
<th>Dimension</th>
<th>Actual Mean</th>
<th>C-means Mean</th>
<th>Gaussian Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setosa</td>
<td>Sepal length</td>
<td>5.006</td>
<td>5.004</td>
<td>5.006</td>
</tr>
<tr>
<td></td>
<td>Sepal width</td>
<td>3.428</td>
<td>3.403</td>
<td>3.418</td>
</tr>
<tr>
<td></td>
<td>Petal length</td>
<td>1.462</td>
<td>1.485</td>
<td>1.464</td>
</tr>
<tr>
<td></td>
<td>Petal width</td>
<td>0.246</td>
<td>0.252</td>
<td>0.244</td>
</tr>
<tr>
<td>Versicolor</td>
<td>Sepal length</td>
<td>5.936</td>
<td>5.889</td>
<td>5.943</td>
</tr>
<tr>
<td></td>
<td>Sepal width</td>
<td>2.770</td>
<td>2.761</td>
<td>2.771</td>
</tr>
<tr>
<td></td>
<td>Petal length</td>
<td>4.260</td>
<td>4.364</td>
<td>4.262</td>
</tr>
<tr>
<td></td>
<td>Petal width</td>
<td>1.326</td>
<td>1.397</td>
<td>1.326</td>
</tr>
<tr>
<td>Virginica</td>
<td>Sepal length</td>
<td>6.588</td>
<td>6.775</td>
<td>6.577</td>
</tr>
<tr>
<td></td>
<td>Sepal width</td>
<td>2.974</td>
<td>3.052</td>
<td>2.971</td>
</tr>
<tr>
<td></td>
<td>Petal length</td>
<td>5.552</td>
<td>5.647</td>
<td>5.542</td>
</tr>
<tr>
<td></td>
<td>Petal width</td>
<td>2.026</td>
<td>2.054</td>
<td>2.022</td>
</tr>
</tbody>
</table>

Mean Square Error: 0.00E+00 5.86E-03 3.36E-05

The same technique used by the MATLAB `gmdistribution.fit` function. The solution with the highest likelihood of the 10 runs is displayed in the tables. The C-means solution was nearly identical across all 10 runs. The Gaussian results varied quite significantly; however, the solution with the highest likelihood was actually the closest to the ideal clustering.

Both algorithms actually compute a fuzzy result whose statistics more accurately match the real data rather than a hard clustering of the data. However, fuzzy results are more difficult to interpret. If a hard classification of the clustering results is performed, where data points are assigned to the cluster with the highest membership value, then 134 out of 150 of the data points (89.3%) are properly classified with C-means. The improperly classified data points are a result of the heuristics of the clustering algorithm itself, not the implementation. The implementation in this thesis agrees identically with the MATLAB Fuzzy C-means `fcm` function results. Both had the same 16 points improperly classified (which are all in the two linearly inseparable iris species) and identical means up to 4 significant digits. The Gaussian mixture model properly classified 147 out of 150 samples (98%).

5.7 Verification - Synthetic Data

This section verifies that the clustering algorithms are functioning properly and assesses the error of the GPU and CPU implementations. Synthetic Gaussian data was generated using the free open-source `mvtnorm` library of the R statistical language. This is similar to the `mvnrnd` function in MATLAB. Both generate a matrix of normally distributed values based on a user-supplied mean vector, covariance matrix, and desired number of vectors. Multiple random clusters are generated.
with different parameters and then concatenated into a single file. The programs developed in this thesis then attempt to locate the clusters.

The first test had two 16 dimensional clusters, each with $2^{16}$ vectors. The clusters only overlap in a couple of dimensions and are distinct (no overlap) in most of the dimensions making it rather simple to cluster. The second cluster has twice the variance of the first cluster. This test focuses on the precision of the algorithms and the difference in accuracy between the GPU and CPU implementations. Table 5.15 shows the clustering results using C-means. Both implementations were using single-precision (32-bit) floating point numbers, began with the same initial clusters, and had the same convergence threshold of $10^{-6}$. Convergence occurs if the maximum change of all cluster centers is less than the threshold. In this test, both the GPU and CPU versions converged after only 7 iterations since the data does not have significant overlap nor are there a large number of clusters. 100% of all data points had a higher membership value in the proper cluster.

Mean-squared error is calculated between the actual (true) value of the randomly generated data and the estimates. The GPU clusters have an average squared error of $1.25 \times 10^{-5}$ while the CPU clusters have an average error of $6.58 \times 10^{-6}$. There is some loss of precision initially since numbers are only stored in single-precision (despite being generated using double precision). Round-off error occurs with floating point operations and can accumulate to significant values with large summations. Despite all of the same calculations being performed throughout the whole algorithm, the order in which they are performed varies significantly between the GPU and CPU implementations. In the GPU implementation the distance calculations, membership calculations, and cluster center updates are distributed to many different threads which are eventually combined together, whereas the CPU implementation just uses large for loops in a single thread. Operations that are mathematically associative (such as multiplication and addition) are not associative with floating point arithmetic in hardware. Although both hardware architectures use the IEEE floating point standard the hardware implementations are likely very different. All of these factors together can cause two implementations of the same mathematical equations to yield slightly different results.

The same data set was then clustered with the Gaussian mixture model algorithm. Results are in Table 5.16. Again, 100% of all data points were properly classified into the correct cluster. The Gaussian algorithm gets closer estimates to the true values than C-means. This is because the randomly generated data are made up of Gaussian distributions. The mixture model takes into account the variance of the clusters and more accurately determines the memberships and cluster centers. With C-means, the first cluster is skewed slightly toward the second cluster which has a higher variance since it computes membership based solely on distance, rather than both distance and variance. MSE of the Gaussian result is on the order of $10^{-13}$ for the GPU and $10^{-9}$ for the CPU using single precision. Although both errors are relatively insignificant, it is interesting that the CPU version was less accurate by a couple orders of magnitude with single precision floats. It is possible that differences in cumulative error or rounding resulted in convergence to a slightly
Table 5.15: Synthetic Data Test 1: C-means

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Dimension</th>
<th>Actual</th>
<th>GPU Estimate</th>
<th>CPU Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1</td>
<td>V01</td>
<td>0.996633</td>
<td>1.004735</td>
<td>1.005276</td>
</tr>
<tr>
<td></td>
<td>V02</td>
<td>1.998676</td>
<td>2.005714</td>
<td>2.006719</td>
</tr>
<tr>
<td></td>
<td>V03</td>
<td>2.999202</td>
<td>3.00516</td>
<td>3.005309</td>
</tr>
<tr>
<td></td>
<td>V04</td>
<td>4.002882</td>
<td>4.007789</td>
<td>4.008307</td>
</tr>
<tr>
<td></td>
<td>V05</td>
<td>5.000341</td>
<td>5.004118</td>
<td>5.00164</td>
</tr>
<tr>
<td></td>
<td>V06</td>
<td>6.00168</td>
<td>6.004381</td>
<td>6.002605</td>
</tr>
<tr>
<td></td>
<td>V07</td>
<td>6.99318</td>
<td>6.994776</td>
<td>6.993715</td>
</tr>
<tr>
<td></td>
<td>V08</td>
<td>7.997636</td>
<td>7.998159</td>
<td>7.997926</td>
</tr>
<tr>
<td></td>
<td>V09</td>
<td>9.00073</td>
<td>9.000144</td>
<td>9.000255</td>
</tr>
<tr>
<td></td>
<td>V10</td>
<td>9.994983</td>
<td>9.99337</td>
<td>9.994564</td>
</tr>
<tr>
<td></td>
<td>V11</td>
<td>11.00116</td>
<td>10.9984</td>
<td>11.00069</td>
</tr>
<tr>
<td></td>
<td>V12</td>
<td>11.99345</td>
<td>11.98961</td>
<td>11.99299</td>
</tr>
<tr>
<td></td>
<td>V13</td>
<td>13.00081</td>
<td>12.996</td>
<td>13.00049</td>
</tr>
<tr>
<td></td>
<td>V14</td>
<td>14.00525</td>
<td>13.9993</td>
<td>14.00493</td>
</tr>
<tr>
<td></td>
<td>V15</td>
<td>15.00088</td>
<td>14.9938</td>
<td>15.00046</td>
</tr>
<tr>
<td></td>
<td>V16</td>
<td>16.00156</td>
<td>15.99344</td>
<td>16.00119</td>
</tr>
<tr>
<td>Cluster 2</td>
<td>V01</td>
<td>16.00116</td>
<td>16.00106</td>
<td>16.001</td>
</tr>
<tr>
<td></td>
<td>V02</td>
<td>15.00765</td>
<td>15.0076</td>
<td>15.0076</td>
</tr>
<tr>
<td></td>
<td>V03</td>
<td>13.98912</td>
<td>13.98897</td>
<td>13.98901</td>
</tr>
<tr>
<td></td>
<td>V04</td>
<td>13.01218</td>
<td>13.01208</td>
<td>13.01198</td>
</tr>
<tr>
<td></td>
<td>V05</td>
<td>11.99667</td>
<td>11.99657</td>
<td>11.9965</td>
</tr>
<tr>
<td></td>
<td>V06</td>
<td>10.99536</td>
<td>10.99523</td>
<td>10.99515</td>
</tr>
<tr>
<td></td>
<td>V07</td>
<td>9.994081</td>
<td>9.99411</td>
<td>9.99419</td>
</tr>
<tr>
<td></td>
<td>V08</td>
<td>8.990774</td>
<td>8.990702</td>
<td>8.990644</td>
</tr>
<tr>
<td></td>
<td>V09</td>
<td>8.004226</td>
<td>8.004307</td>
<td>8.004337</td>
</tr>
<tr>
<td></td>
<td>V10</td>
<td>7.003521</td>
<td>7.003541</td>
<td>7.003566</td>
</tr>
<tr>
<td></td>
<td>V11</td>
<td>5.99651</td>
<td>5.996633</td>
<td>5.996632</td>
</tr>
<tr>
<td></td>
<td>V12</td>
<td>4.999828</td>
<td>4.999841</td>
<td>4.999872</td>
</tr>
<tr>
<td></td>
<td>V13</td>
<td>4.007138</td>
<td>4.007238</td>
<td>4.007246</td>
</tr>
<tr>
<td></td>
<td>V14</td>
<td>2.996025</td>
<td>2.996013</td>
<td>2.99602</td>
</tr>
<tr>
<td></td>
<td>V15</td>
<td>2.00032</td>
<td>2.000473</td>
<td>2.000476</td>
</tr>
<tr>
<td></td>
<td>V16</td>
<td>0.995601</td>
<td>0.995776</td>
<td>0.995775</td>
</tr>
</tbody>
</table>

Mean Square Error 0.00E+00 1.25E-05 6.58E-06
different local minima. With double precision on the CPU the accuracy is significantly higher, on the order of $10^{-24}$. 

\textit{double} has a precision of $1.0/2^{53}$ (about 16 decimal digits) whereas \textit{float} only has a precision of $1.0/2^{24}$ (between 7 and 8 decimal digits). Therefore doubling the precision should (and does) approximately square the MSE value. Nevertheless, the single precision arithmetic available on the GPU appears to be suitable for the clustering algorithms and the computations do not become unstable.

Table 5.16: Synthetic Data Test 1: Gaussians

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Dimension</th>
<th>Actual</th>
<th>GPU</th>
<th>CPU float</th>
<th>CPU double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1</td>
<td>V01</td>
<td>0.996633</td>
<td>0.996633</td>
<td>0.996635</td>
<td>0.996633</td>
</tr>
<tr>
<td></td>
<td>V02</td>
<td>1.998676</td>
<td>1.998676</td>
<td>1.998675</td>
<td>1.998676</td>
</tr>
<tr>
<td></td>
<td>V03</td>
<td>2.999202</td>
<td>2.999202</td>
<td>2.999204</td>
<td>2.999202</td>
</tr>
<tr>
<td></td>
<td>V04</td>
<td>4.002882</td>
<td>4.002882</td>
<td>4.002881</td>
<td>4.002882</td>
</tr>
<tr>
<td></td>
<td>V05</td>
<td>5.000341</td>
<td>5.000341</td>
<td>5.000333</td>
<td>5.000341</td>
</tr>
<tr>
<td></td>
<td>V06</td>
<td>6.00168</td>
<td>6.001679</td>
<td>6.001652</td>
<td>6.00168</td>
</tr>
<tr>
<td></td>
<td>V07</td>
<td>6.99318</td>
<td>6.99318</td>
<td>6.993167</td>
<td>6.99318</td>
</tr>
<tr>
<td></td>
<td>V08</td>
<td>7.997636</td>
<td>7.997636</td>
<td>7.997647</td>
<td>7.997636</td>
</tr>
<tr>
<td></td>
<td>V11</td>
<td>11.00116</td>
<td>11.00116</td>
<td>11.00118</td>
<td>11.00116</td>
</tr>
<tr>
<td></td>
<td>V12</td>
<td>11.99345</td>
<td>11.99345</td>
<td>11.99344</td>
<td>11.99345</td>
</tr>
<tr>
<td></td>
<td>V15</td>
<td>15.00088</td>
<td>15.00088</td>
<td>15.00082</td>
<td>15.00088</td>
</tr>
</tbody>
</table>

| Cluster 2 | V01  | 16.00116 | 16.00116 | 16.00115 | 16.00116 |
|           | V02  | 15.00765 | 15.00765 | 15.00769 | 15.00765 |
|           | V05  | 11.99667 | 11.99668 | 11.99656 | 11.99667 |
|           | V06  | 10.99536 | 10.99536 | 10.99528 | 10.99536 |
|           | V08  | 8.990774 | 8.990774 | 8.990699 | 8.990774 |
|           | V09  | 8.004226 | 8.004227 | 8.004227 | 8.004226 |
|           | V10  | 7.003521 | 7.003521 | 7.003518 | 7.003521 |
|           | V11  | 5.99651  | 5.99651  | 5.996506 | 5.99651  |
|           | V12  | 4.999828 | 4.999827 | 4.999831 | 4.999828 |
|           | V13  | 4.007138 | 4.007138 | 4.007144 | 4.007138 |
|           | V14  | 2.996025 | 2.996025 | 2.996025 | 2.996025 |
|           | V15  | 2.00032  | 2.00032  | 2.000319 | 2.00032  |
|           | V16  | 0.995601 | 0.995601 | 0.9956  | 0.995601 |

| Mean Square Error | 0.00E+00 | 3.51E-13 | 1.58E-09 | 7.84E-24 |

A second set of test data was constructed with 6 one-dimensional Gaussians. This test has clusters with more overlap (and no extra dimensions to provide linear separation), varying proportions,
and different variances. The dataset contained a total of 100,000 points. Figure 5.37 shows a plot of the data mixture. The red line is the whole mixture, and the blue lines are the individual Gaussians that when added together form the entire data set. Table 5.17 shows the parameters used to generate the data. The actual parameters of the randomly generated data are accurate to within 2 decimal places of the expected values.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Proportion</th>
<th>Mean</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.40</td>
<td>-10.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.15</td>
<td>-5.00</td>
<td>4.00</td>
</tr>
<tr>
<td>3</td>
<td>0.15</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>0.20</td>
<td>1.00</td>
<td>0.25</td>
</tr>
<tr>
<td>5</td>
<td>0.05</td>
<td>5.00</td>
<td>2.25</td>
</tr>
<tr>
<td>6</td>
<td>0.05</td>
<td>10.00</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 5.17: Test #2 Dataset

Figure 5.37: Test 2 Gaussian Mixture Data

Table 5.18 shows the clustering results using C-means with six clusters. C-means was not able to accurately cluster all of the Gaussians; however the results agreed with the MATLAB \texttt{fcm} implementation of C-means. Table 5.19 shows the clustering results using Expectation Maximization with Gaussians. Using the same approach as discussed for the Iris data, the clustering was performed 10 times and the result with the highest likelihood was chosen. All 6 clusters were found in the best solution; however many of the 10 clusterings did not properly find the underlying clusters. The most common problem was that the two closest clusters (with means at 0.00 and 1.00) became a single cluster, and the large cluster centered at -10.00 with a proportion of 40% was split into two
overlapping clusters. This result is not surprising considering the overlap of the clusters and the lack of other dimensions to help distinguish clusters. If two clusters are randomly initialized closely together then both of these clusters will likely converge to the same Gaussian, (effectively describing a single larger distribution with two smaller ones). Starting with a larger number of clusters and merging the closest Gaussians together with agglomerative hierarchical clustering produced much more consistent results, (at the expense of longer execution time). When starting with 10 clusters, 8 out of 10 runs found the proper 6 clusters. When starting with 20 clusters, all 10 clusterings found the same 6 clusters.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Proportion</th>
<th>Center</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.222001</td>
<td>-10.73</td>
</tr>
<tr>
<td>2</td>
<td>0.206789</td>
<td>-9.02</td>
</tr>
<tr>
<td>3</td>
<td>0.105699</td>
<td>-4.84</td>
</tr>
<tr>
<td>4</td>
<td>0.165537</td>
<td>-0.31</td>
</tr>
<tr>
<td>5</td>
<td>0.232082</td>
<td>1.34</td>
</tr>
<tr>
<td>6</td>
<td>0.067891</td>
<td>9.49</td>
</tr>
</tbody>
</table>

Table 5.19: Test #2 EM Gaussian Clustering Summary

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Proportion</th>
<th>Mean</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.400777</td>
<td>-9.99983</td>
<td>1.010982</td>
</tr>
<tr>
<td>2</td>
<td>0.147426</td>
<td>-5.05974</td>
<td>3.787570</td>
</tr>
<tr>
<td>3</td>
<td>0.146837</td>
<td>-0.04929</td>
<td>1.019124</td>
</tr>
<tr>
<td>4</td>
<td>0.205379</td>
<td>0.987505</td>
<td>0.261684</td>
</tr>
<tr>
<td>5</td>
<td>0.049552</td>
<td>5.049315</td>
<td>2.170046</td>
</tr>
<tr>
<td>6</td>
<td>0.050028</td>
<td>10.00116</td>
<td>0.250567</td>
</tr>
</tbody>
</table>

Figure 5.38 shows the MDL (Rissanen) scores for the different cluster configurations. The hierarchical clustering iteratively merges from 20 Gaussians down to a single Gaussian. According to MDL, the minimum score should be the best solution. In this case, the 6 cluster solution has the lowest (best) MDL score. With less than 6 clusters there is a very sharp increase in the MDL score, which indicates that a lot of the information in the data set is not well described by the solution. Over-fitting with too many clusters still describes the data well, but the information criterion adds a penalty to try to compensate for over-fitting. In this test MDL with a Gaussian mixture model worked well since the underlying data was in fact synthetically generated Gaussian data.
5.8 Flow Cytometry Data

The field of flow cytometry has a lack of labeled data sets that can be used for verification. A labeled data set is one that has additional information (labels) that classify the vectors into different clusters — in other words, the ground truth is known and it gives algorithm developers a benchmark to compare results against. This section introduces a real flow cytometry data set created by the Flowgating group at the University of Rochester Center for Vaccine Biology and Immunology and displays some clustering results. The idea behind the experiment was to create data sets that contain real-world non-trivial data, but with some known factors that can be used to compare clustering results. A combination of human blood cells and mouse blood cells is used. The mouse blood has characteristics that are very similar to human blood, but different antigens can be used that will bind to only mouse or human cells. This creates data with real-world characteristics but contains dimensions that can be used to distinguish the classification of the events (cells).

The data set contains 22 dimensions. The first six are scatter data, which essentially measure physical characteristics of the cells. The remaining variables are fluorescent. Table 5.20 summarizes the variables (dimensions) in the data set. If \( m \) prefaces a fluorescent label, it indicates a mouse specific variable. Similarly an \( h \) indicates a human variable. The labels are not absolute, but generally only a mouse blood cell should exhibit a high fluorescence value for a mouse variable, similarly only human cells should exhibit large fluorescence values for human variables. However, an event that is actually a human cell can exhibit either a negative (low fluorescence value) or positive (high value)
Table 5.20: Human/Mouse Flow Cytometry Data Variables

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Type</th>
<th>Laser</th>
<th>Description / Fluorescent Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Scatter</td>
<td>FSC-A</td>
<td>Forward Scatter Area</td>
</tr>
<tr>
<td>2</td>
<td>Scatter</td>
<td>FSC-H</td>
<td>Forward Scatter Height</td>
</tr>
<tr>
<td>3</td>
<td>Scatter</td>
<td>FSC-W</td>
<td>Forward Scatter Width</td>
</tr>
<tr>
<td>4</td>
<td>Scatter</td>
<td>SSC-A</td>
<td>Side Scatter Area</td>
</tr>
<tr>
<td>5</td>
<td>Scatter</td>
<td>SSC-H</td>
<td>Side Scatter Height</td>
</tr>
<tr>
<td>6</td>
<td>Scatter</td>
<td>SSC-W</td>
<td>Side Scatter Width</td>
</tr>
<tr>
<td>7</td>
<td>Fluorescent</td>
<td>Blue B 515/20-A</td>
<td>mCD8a FITC</td>
</tr>
<tr>
<td>8</td>
<td>Fluorescent</td>
<td>Blue A 710/50-A</td>
<td>mCD452 PerCP-Cy55</td>
</tr>
<tr>
<td>9</td>
<td>Fluorescent</td>
<td>Violet H 450/50-A</td>
<td>hCD45 AF405</td>
</tr>
<tr>
<td>10</td>
<td>Fluorescent</td>
<td>Violet G 550/40-A</td>
<td>Live/Dead Aqua</td>
</tr>
<tr>
<td>11</td>
<td>Fluorescent</td>
<td>Violet D 605/40-A</td>
<td>hCD3 QD605</td>
</tr>
<tr>
<td>12</td>
<td>Fluorescent</td>
<td>Violet C 640/60-A</td>
<td>hCD19 QD655</td>
</tr>
<tr>
<td>13</td>
<td>Fluorescent</td>
<td>Violet A 780/60-A</td>
<td>hCD14 QD800</td>
</tr>
<tr>
<td>14</td>
<td>Fluorescent</td>
<td>Red C 660/20-A</td>
<td>mF4/80 APC</td>
</tr>
<tr>
<td>15</td>
<td>Fluorescent</td>
<td>Red B 710/50-A</td>
<td>mCD4 AF700</td>
</tr>
<tr>
<td>16</td>
<td>Fluorescent</td>
<td>Red A 780/60-A</td>
<td>hCD4 APC-eF780</td>
</tr>
<tr>
<td>17</td>
<td>Fluorescent</td>
<td>Green E 575/25-A</td>
<td>mNK11 PE</td>
</tr>
<tr>
<td>18</td>
<td>Fluorescent</td>
<td>Green D 610/20-A</td>
<td>hCD8 PE-TR</td>
</tr>
<tr>
<td>19</td>
<td>Fluorescent</td>
<td>Green C 660/40-A</td>
<td>mCD19 PE-Cy5</td>
</tr>
<tr>
<td>20</td>
<td>Fluorescent</td>
<td>Green B 710/50-A</td>
<td>hCD56 PE-Cy55</td>
</tr>
<tr>
<td>21</td>
<td>Fluorescent</td>
<td>Green A 780/40-A</td>
<td>mCD3 PE-Cy7</td>
</tr>
<tr>
<td>22</td>
<td>Time</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

for a human marker. For example, the *hCD3* is a marker that gives a positive response with T-cells, which is a subset of the human white blood cells. Negative and positive responses should not be confused with negative and positive integers. All measured values should be positive integers, but some flow cytometers subtract a small constant from the data. Transformation and compensation of the data can also produce negative values.

The fluorescent variables has been compensated, transformed with a variation of the Logicle transformation [27], and rescaled back to the original $2^{18}$ range as described in Workflow section of Chapter 2 before performing any clustering or plotting. Figure 5.39 shows a common flow cytometry density plot between the forward and side scatter variables. The plot is formed by computing a fine-grained 2D histogram and then mapping the log of the counts in the bins to colors in a heat map. Red indicates a high density of events while blue indicates low density.

Figure 5.40 shows plots of two fluorescent variables — one human and one mouse. The plot on the top-left has only human cells. The majority of the human cells has a negative response on the X-axis variable (mCD4) and either a negative or positive response on the Y-axis (hCD45). Conversely, the plot on the top-right has only mouse cells. The bottom of Figure 5.40 is the synthetic (electronic) mixture of the cells. By combining the cell data electronically it is possible to cluster the data and then compare the composition of human and mouse cells in the clusters. Ideally the cell populations
Although this experiment focuses on distinguishing only two types of cells (human and mouse), it is necessary to use more than two clusters for the analysis. The data has complex shapes, background noise, dead cells, doublets (multiple cells physically stuck together that go through the cytometer as a single event), cells particles, and other impurities. It is not possible to accurately describe all of this data with only two spherical (C-means) or Gaussian clusters. Figure 5.41 shows a plot of the number of cells that were improperly clustered (a mouse cell in a predominately human cluster or vice versa) as a function of the number of clusters. Adding more clusters allows the human and mouse distinction to be more prevalent, rather than being dominated by other criteria such as dead vs. alive, or small vs. large cells. 64 clusters was chosen as the number of clusters for the following tests.

The first test performed on this synthetic data set used a mixture of 100,000 human cells and 100,000 mouse cells. Clustering was performed with 64 Gaussians. 10 replicates were performed with different random initial starting conditions and the highest likelihood result was chosen. A GTX260 GPU was used and clustering of each replicate took 50 seconds, including I/O and all other
Figure 5.40: Density Plot of an electronic mixture (bottom) of Human blood cells (top-left) and Mouse blood cells (top-right) with hCD45 and mCD4 variables
overheads. 54 of the 64 clusters properly classified 99.9% or more of the cells in the cluster. Only 3 clusters were worse than 90%. These clusters seemed to be Gaussians with low total membership and very high variance that were attempting to describe the background noise and outliers in the data. A total of 99.3% of all 200,000 cells were properly classified in this data set.

Figure 5.42 shows a plot of 16 of the 64 clusters. The dimensions are the same as the density plots shown above in Figure 5.40. Only 4 clusters are shown per subplot (red, green, blue, and purple) to reduce clutter. Note that values with the same color in different plots are not the same clusters. The clustering algorithm produces soft (fuzzy) classifications of cells, but for plotting purposes it is necessary to map the events to distinct (hard) clusters. Events are placed into clusters by treating the membership values as a probability mass function (PMF) and then randomly assigning it a cluster based on the distribution in the PMF. For example, if an event has a 0.1 membership in cluster 1 and 0.9 membership in cluster 2, then there is a 10% probability that it will be assigned in cluster 1 and a 90% probability that it will be assigned to cluster 2 for plotting. This technique produces figures that more accurately represent the soft clustering result than a simple classification based on the maximum membership value, particularly with overlapping clusters.

One could argue that distinguishing mouse cells from human cells is too trivial given so many dimensions that are either mouse-only or human-only. However, in none of the dimensions are all of the human or mouse cells linearly separable from each other. Furthermore, being real data it contains many impurities such as: dead cells, particles/debris, doublets (multiple cells stuck together), and
Figure 5.42: Plot of 16 clusters with a 50% mixture of human and mouse cells: hCD45 vs. mCD4
fluorescent spillover. The results show that the implementation does work on real data from a flow cytometer, and not just synthetically generated Gaussian data.

It is possible to make the distinction between mouse and human cells more challenging by using an unbalanced mixture of cells (therefore more comparable to finding small cell populations in a human-only data set). First a 9% mixture was formed with 10,000 human cells and 100,000 mouse cells. 99.29% human cells, 98.96% mouse cells, and a total of 98.99% of all cells were properly classified. The mixture was then reduced to only 1,000 (0.9%) human cells and 100,000 mouse cells. 62 (6.2%) human cells were improperly classified as mouse cells, and 192 (0.19%) mouse cells were classified as human cells. These results are not surprising considering the aforementioned impurities in the real-world data. The cluster centers are also much more likely to converge to the denser mouse cells rather than the very sparse 1,000 human cells given the way expectation maximization operates.
Chapter 6

Conclusion

• The C-means and Expectation Maximization with Gaussian mixture models clustering algorithms exhibit substantial data parallelism that efficiently maps to many-core GPU architectures using NVIDIA's CUDA framework.

• GPUs allow clustering of large data sets, like those found in flow cytometry, to be practical on a single machine. Clustering takes only a few minutes instead of many hours.

• GPU co-processors and the CUDA framework can be combined with traditional parallel programming techniques, such as OpenMP and MPI, for efficient high performance computing. This provides both low latency for individual data sets and high throughput for large scale analyses.

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 thesis 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 thesis 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 thesis 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 ($4 \times 10^6$) 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.
Chapter 7

Future Work

Multivariate data clustering as a field has existed for many decades; however its application to the field of flow cytometry is relatively new. Clustering flow cytometry data is very challenging for a number of reasons, such as the complexity of the shape of the clusters, significant overlap between clusters, moderately high dimensionality, and the large size of the data sets, to name a few. FlowCAP [45] is a new project attempting to objectively test computational methods for the analysis of flow cytometry data.

This thesis 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 [23] 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 following are other areas of flow cytometry analysis, aside from data clustering, that may benefit from GPU acceleration.

- Pre-conditioning the data, which involves filtering, solving a linear system for compensation, and data transformations
- Statistical inference across multiple data sets
- Visualization of results

The two algorithms in this thesis 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 CUDAtools could be investigated.

One current limitation of the MPI implementation in this thesis 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:

- Remove OpenMP and try an MPI-only implementation.
- Use other emerging GPU programming models besides CUDA such as OpenCL, DirectX DirectCompute, or PGI.
- Comparison of GPU to multi-core CPU implementations of the algorithms, MPI implementations without GPUs.
- 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 UpdateCenters, EM E-step, and EM constants kernels.
Bibliography


