I dedicate this to my parents and my friends.
ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Sanjay Ranka for all the guidance and encouragement provided throughout my masters program. I also thank my thesis committe members, Dr. Alin Dobra, Dr. Jih-Kwon Peir for their help and suggestions.

I would like to thank Dr. Gordon Erlebacher, and the IT support staff of Florida State University for providing access to the machines for benchmarking.

I thank my parents Dr. CP Ravunnikutty, Dr. Geethadevi Ravunnikutty and my brother Mahesh Ravunnikutty for their immense care and support that motivated me during my studies. Finally I thank all of my friends who helped me with the thesis work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>9</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>10</td>
</tr>
<tr>
<td>2 PRELIMINARIES</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Clustering Problem</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Ensemble $K$-means Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Index Based Ensemble $K$-means Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>2.4 Related Work</td>
<td>16</td>
</tr>
<tr>
<td>3 OPENCL DEVICE ARCHITECTURE</td>
<td>17</td>
</tr>
<tr>
<td>4 PARALLEL ENSEMBLE $K$-MEANS ALGORITHM</td>
<td>22</td>
</tr>
<tr>
<td>4.1 Parallelization Issues</td>
<td>22</td>
</tr>
<tr>
<td>4.1.1 Concurrency</td>
<td>22</td>
</tr>
<tr>
<td>4.1.2 Memory Constraints</td>
<td>23</td>
</tr>
<tr>
<td>4.1.3 Synchronization and Atomic Operations</td>
<td>24</td>
</tr>
<tr>
<td>4.1.4 Data Transfer Between Host and Device</td>
<td>24</td>
</tr>
<tr>
<td>5 OPENCL IMPLEMENTATION</td>
<td>25</td>
</tr>
<tr>
<td>5.1 Task Parallelism</td>
<td>25</td>
</tr>
<tr>
<td>5.2 Data Parallelism</td>
<td>28</td>
</tr>
<tr>
<td>5.3 Concatenated Parallelism</td>
<td>30</td>
</tr>
<tr>
<td>6 PARALLEL $K$-MEANS ALGORITHM USING KD-TREE</td>
<td>35</td>
</tr>
<tr>
<td>6.1 Concatenated Parallelism Using KD-tree</td>
<td>37</td>
</tr>
<tr>
<td>6.1.1 Direct Access</td>
<td>37</td>
</tr>
<tr>
<td>6.1.2 Distributed Access</td>
<td>38</td>
</tr>
<tr>
<td>6.1.3 Eliminating Memory Conflicts Using Replication</td>
<td>38</td>
</tr>
<tr>
<td>7 EMPIRICAL EVALUATION</td>
<td>39</td>
</tr>
<tr>
<td>7.1 Datasets</td>
<td>39</td>
</tr>
<tr>
<td>7.2 Benchmark Machines</td>
<td>39</td>
</tr>
<tr>
<td>7.3 Comparison of Three Clustering Approaches on a 32-core CPU</td>
<td>39</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1 Parallelization Methodologies</td>
<td>23</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>CPU vs GPU Architecture</td>
<td>17</td>
</tr>
<tr>
<td>3-2</td>
<td>Coalesced Global Memory Access</td>
<td>18</td>
</tr>
<tr>
<td>3-3</td>
<td>Non-Coalesced Global Memory Access</td>
<td>19</td>
</tr>
<tr>
<td>3-4</td>
<td>Bank Conflicts in Local Memory</td>
<td>19</td>
</tr>
<tr>
<td>3-5</td>
<td>OpenCL Compute Device</td>
<td>19</td>
</tr>
<tr>
<td>5-1</td>
<td>Task Parallelism</td>
<td>26</td>
</tr>
<tr>
<td>5-2</td>
<td>Data Parallelism</td>
<td>29</td>
</tr>
<tr>
<td>5-3</td>
<td>Concatenated Parallelism</td>
<td>31</td>
</tr>
<tr>
<td>5-4</td>
<td>High Contention Local Memory Atomic Update</td>
<td>32</td>
</tr>
<tr>
<td>5-5</td>
<td>Reduced Contention Local Memory Atomic Update</td>
<td>32</td>
</tr>
<tr>
<td>6-1</td>
<td>KD-tree with Data Points and Mapped Centroids on Leaf Nodes</td>
<td>35</td>
</tr>
<tr>
<td>6-2</td>
<td>KD Tree: Direct Access</td>
<td>37</td>
</tr>
<tr>
<td>6-3</td>
<td>KD Tree: Distributed Mapping</td>
<td>38</td>
</tr>
<tr>
<td>7-1</td>
<td>Comparison of Three Clustering Approaches on a 32 core CPU</td>
<td>40</td>
</tr>
<tr>
<td>7-2</td>
<td>Comparison of Three Clustering Approaches on FERMI</td>
<td>41</td>
</tr>
<tr>
<td>7-3</td>
<td>Performance Comparison of Different Hardware</td>
<td>41</td>
</tr>
<tr>
<td>7-4</td>
<td>Comparison of the Three KD-tree Algorithms</td>
<td>42</td>
</tr>
<tr>
<td>7-5</td>
<td>KD-tree vs Direct Implementation of Concatenated Parallelism</td>
<td>43</td>
</tr>
<tr>
<td>7-6</td>
<td>KD-tree vs Direct Implementation for Varying Dimensions</td>
<td>43</td>
</tr>
</tbody>
</table>
Clustering involves partitioning a set of objects into subsets called clusters so that objects in the same cluster are similar according to some metric. Clustering is widely used in many fields like machine learning, data mining, pattern recognition and bioinformatics. K-means algorithm is the most popular algorithm used for clustering which uses distance as the similarity metric. K-means algorithm works by choosing a set of K random objects (model) from the data set and performs distance computation. Ensemble K-means chooses multiple models with the goal of refinement and faster convergence.

The focus of the thesis is to investigate different approaches to parallelize the ensemble based K-means algorithms on modern many core hardware. The many core hardware involves GPUs and CPUs backed by the OpenCL software stack. This thesis also presents novel implementations that exploit the immense computing power of the hardware by minimizing the amount of data access.
A number of traditional machine learning and data mining algorithms generate multiple models using the same dataset. These models are generated by choosing different initializations that are generated randomly or using a systematic method. These models are then used to generate a single model or a committee of models. For example, in most clustering algorithms, multiple runs with different randomly generated initializations are processed and the one with the minimum mean square error or number of clusters or a combination of both may be chosen. Similarly, for classification algorithms, multiple models may be built, where each model predicts the class of a given input. A majority based output is then used to predict the class for a new sample. In these scenarios, the key challenge is building multiple models. The computational intensiveness of the algorithms makes it a significant challenge especially for large data sets. The thesis explores the use of multi-core hardware for performing such computing to generate multiple models in an ensemble. We assume that the models in the ensemble are independently generated. Although, our methods can be easily extended when a sequence of ensembles are generated, in which each ensemble consist of multiple models. The latter is typical of genetic algorithms and other evolutionary approaches. There are two broad ways of using the multi core processor for this purpose:

1. Task parallelism: Generate each model separately on each of the cores of the processor. This prevents the amount of communication between each of the cores but has the potential for load imbalance if the different models need different amounts of time.

2. Data parallelism: Use multiple (or all) the cores to utilize the data parallelism that is present in many of these model generation algorithms. This has the advantage of achieving good balance but may generate communication between the cores.

We show both these approaches suffer from the limitation that they require multiple passes of the entire dataset. This requires extensive data traffic from the
global memory to the local memory of the cores. Given the two to three orders of magnitude performance difference between global and local memory and the limited overall bandwidth of the bus, the additional overhead generated by multiple passes can deteriorate the performance significantly. We present a novel approach called concatenated parallelism that effectively utilizes task as well as data parallelism to reduce the number of passes of data between local memory and global memory.

We demonstrate the effectiveness of our approach using \( K \)-means clustering. \( K \)-means clustering partitions data records into related groups based on a model of randomly chosen features without prior knowledge of the group definitions. It is a classic example of unsupervised machine learning. Multiple models are typically generated by using different random initialization or by varying the number of clusters. For generation of a single model using the \( K \)-means algorithm, a random set of \( K \) records are chosen from the given records (data points). These \( K \) records form the initial cluster centroids. Each data point is compared against the \( K \) cluster centroids and is assigned to the closest centroid. In Ensemble clustering, instead of single prototype of \( K \) centroids, \( M \) models or prototypes each of \( K \) centroids are chosen. Thus data points are compared with \( K \times M \) cluster centroids and the best model or prototype is selected.

We consider two approaches for building the models. In a simple approach, in each iteration, all the centroids for each model are updated using the entire dataset. In a more complex approach, the dataset is stored as a KD tree, where the leaf nodes correspond to a subset of records. A preprocessing step determines which centroids can be potential nearest neighbors for all the records corresponding to a given leaf node. This approach has the benefit of reducing the total number of comparisons to determine the nearest neighbor especially when the number of attributes is small.

We benchmark our code on multi-core CPU and GPU hardware from multiple vendors. Our experimental results show that our approach has several important benefits:
1. It significantly reduces the amount of the data access costs by limiting the number and amount data that is transferred from the global memory to any of the cores.

2. It effectively utilizes the SIMT (Single Instruction Multiple Thread) nature of the underlying hardware by exploiting the data parallelism.

   Additionally, we propose several approaches to minimize the update contention that is generated by the SIMT nature of each core. We reduce the global memory access latency by using local memory and memory coalescing. Our experiment also incorporates performance enhancing techniques like loop unrolling, reducing local memory bank conflicts, optimized register usage and using constant memory.

   The key outcome is that the resultant code amortizes a larger number of computations per data access and is an order of magnitude faster than straightforward parallelization techniques. We believe that this Multiple Instruction Single Data (MISD) data is extremely important to derive performance on GPU for a large class of data intensive and data mining operations. Although our techniques have been described in the context of ensemble clustering, they are quite general and should be applicable to a variety of ensemble based applications.

   The thesis is organized as follows: Chapter 2 gives a background of clustering and ensemble clustering problems. It also discusses in detail the $K$-means algorithm used for clustering. Chapter 3 describes the OpenCL device architecture. The chapter also details the various performance tweaks while writing OpenCL code. Chapter 4 starts by describing the issues to be addressed for parallelizing any algorithm on OpenCL hardware. In the end, we discuss how these issues are handled in the parallelization of $K$-means algorithm. Chapter 5 presents three methodologies to parallelize $K$-means algorithm on OpenCL hardware. We also discuss the Asymptotic computation and data access complexities for each of these methodologies. Chapter 6 discusses an enhancement over the $K$-means algorithm using KD-tree. We discuss three methodologies for $K$-means clustering using a KD-tree. Chapter 7 presents the experimental results. We compare the performance of methodologies described in
Chapter 5 and Chapter 6. Chapter 8 summarizes the thesis and discusses the directions for future work.
2.1 Clustering Problem

Clustering or cluster analysis problem involves finding groups of records from a huge collection such that each record in a group will be more similar to others in the group and different from the records in other groups. Clustering uses distance between records. The distance is typically calculated as a linear combination of the distance along each of the many attributes. Thus, finding an appropriate distance measure becomes an important step in clustering. The distance measure decides the shape of clusters because two records can be close to one another in one dimension where as they will be far according to another dimension. We choose the Euclidean distance metric for our work. We also assume that all attributes or dimension are equally weighted. Our algorithms and approaches can be suitably generalized to other distance measures.

Lloyd’s algorithm, also known as K Means algorithm is one of the most popular algorithms used in clustering. The algorithm partitions $N$ records to K clusters. In $K$ - means, a random set of $K$ records are chosen from the given records (data points). These $K$ records form the initial cluster centroids. Each data point is compared against the $K$ cluster centroids and is assigned to the closest centroid.

Given a set of $N$ records $(n_1, n_2, \cdots n_n)$, where each record is a $d$-dimensional vector, the $K$-means clustering partitions the $N$ records into $K$ clusters $S_1, S_2, \cdots, S_k$ where $K < N$, such that intra cluster distance is minimized and inter cluster distance is maximized. The number of clusters is fixed in $K$-means clustering and must be specified. Let the initial centroids $w_1, w_2, \cdots, w_k$ be initialized to one of the $N$ records. The quality of the clustering is determined by the following error function.

$$E = \sum_{i=1}^{k} \sum_{n_i \in C_j} \| n_i - w_j \|^2$$  \hspace{1cm} (2–1)
where \( C_j \) is the \( j^{th} \) cluster whose value is a disjoint subset of input patterns.

\( K \) means algorithm works iteratively on a given set of \( K \) clusters. Each iteration consists of two steps:

- Every data record is compared with the \( K \) centroids and is associated with the closest centroid creating \( K \) clusters.
- The new sets of centroids are determined as the mean of the points in the cluster created in the previous step.

The algorithm repeats until the centroids do not change or when the error reaches a threshold value. The computational complexity of algorithm is \( O(NKdl) \) where \( l \) is the number of iterations.

### 2.2 Ensemble \( K \)-means Algorithm

Ensemble \( K \)-means algorithm is an extension to \( K \) means algorithm with the goal of refinement of results and provide faster convergence. Instead of a single model with \( K \) centroids, we choose \( M \) models of \( K \) cluster centroids. Each iteration of \( K \) means operates on \( K \times M \) centroids. Each data point compared with the centroids of each model is assigned to the closest centroid in each of the \( M \) models. The computational complexity of the algorithm is \( O(NKdMI) \).

### 2.3 Index Based Ensemble \( K \)-means Algorithm

In direct ensemble \( K \)-means clustering algorithm, each centroid in a model is compared with all the data points. This requires time proportional to the product of number of models and number of clusters. As an enhancement to this, we can use index based data structures to reduce the computations performed. The data structure can partition the data records so that centroids can identify a subset of records to which they are close instead of scanning the whole data records. In the thesis, we use KD-tree as the indexing data structure. A KD-tree is created for data points with each leaf node containing a set of data points. In KD-tree each internal node is an axis orthogonal hyper plane which divides the set of points in the space into two hyper rectangles. Once the KD-tree creation is done, all the data points will be in the leaf nodes. Each centroid
in a model traverses the KD-tree and gets mapped to the data points in the leaf nodes. KD-tree based ensemble clustering is explained in detail in Chapter 6.

### 2.4 Related Work

The $K$-means clustering method has been implemented on GPU. Before the introduction of NVIDIA CUDA framework, clustering problem has been implemented on GPU by mapping to texture and shader used for graphics computation [2, 8, 14]. These implementations use the OpenGL APIs. After the introduction of NVIDIA CUDA framework, there are several GPU implementations of $K$-means clustering problem. Chang et.al. [3, 4] computed pairwise Euclidean distance, Manhattan distance and Pearson correlation coefficient with speed-up over CPU. This was extended to the first complete hierarchical clustering on GPU [5]. Cao et.al. [2] describes the GPU clustering method for data sets that are too large to fit in the GPU’s global memory. In the CUDA clustering implementation done by Che et.al. [6], CPU computes the new cluster centroids which necessitates the transfer of data between CPU and GPU in each clustering iteration. Li et.al. [10] describes the CUDA implementation of clustering using registers for low dimensional data and shared memory for higher dimensions. Fang et.al. [7] proposes a GPU based clustering approach using a bitmap to speed up the counting in each iteration. This approach does not scale well for ensemble clustering as bitmap increases the memory foot print. The algorithms described in the thesis does not require data transfers between successful clustering iterations. The implementations are done in OpenCL which ensures the portability to any heterogeneous hardware backed up by an OpenCL device driver.
In the thesis, we use GPUs and multi core CPUs as OpenCL devices. GPU is a dedicated computing device to address problems that have high arithmetic intensity i.e. high ratio of arithmetic operations to memory operations. Each of the GPU cores has a SIMT (Single Instruction Multiple Thread) architecture. The core of a GPU has a small cache and limited flow of control - both of these take up most of the transistors in a CPU whereas in a GPU, more transistors are used up for computing cores. Figure 3-1 shows a high level architecture view of CPU and GPU.

Parallel portions of an application are expressed as device kernels which run on many threads. GPU threads are extremely lightweight and the creation overhead is very little. The scheduling is done by the hardware unlike the operating system in a CPU. A typical GPU needs hundreds of threads for full utilization of hardware whereas CPU can be saturated with only a few threads.

ATI (Owned by AMD) and NVIDIA are the leading vendors that provide GPUs with general purpose computing capabilities. NVIDIA provides the Compute Unified Device Architecture (CUDA) Framework [11] with a new parallel programming model and Instruction Set Architecture to leverage the capabilities of the massive parallel computing hardware in NVIDIA GPUs. CUDA framework provides an extended C application programming interface and a runtime library which enables the access and

![Figure 3-1. CPU vs GPU Architecture](image-url)
control of devices from the host. Similarly AMD provides ATI stream technology that enable AMD CPUs and GPUs to accelerate applications.

OpenCL (Open Computing Language) [9] is an open royalty-free standard to write parallel programs that execute across heterogeneous computing environment consisting of CPUs, GPUs. OpenCL framework provides a ISO C99 based language for writing portable code that executes on heterogeneous platforms. NVIDIA provide OpenCL drivers for their GPUs and AMD provides for the CPUs and ATI GPUs.

An OpenCL device is identified as a collection of compute units. Each compute unit can contain many Processing Elements (PEs). OpenCL program executes in two parts: Kernels that execute on OpenCL devices and a host program that executes on the host. The instance of the kernel that is executing on a processing is called work-item. Work-items are organized into work groups. Work-items in a work-group executes the same code in SIMD fashion on all the processing elements of a compute unit. Each work item in a work group has a unique ID and the work-groups has a global unique ID. In NVIDIA CUDA framework, work-items are identified as threads and work-group as blocks. The application running on the hosts uses OpenCL APIs to control the execution of kernels on devices. OpenCL provides a command queue interface to coordinate execution of kernels on the devices from the host. The host places kernel execution, memory transfer and synchronization commands into the command queue and are then scheduled to execute on the devices.
Figure 3-3. Non-Coalesced Global Memory Access

Figure 3-4. Bank Conflicts in Local Memory

Figure 3-5. OpenCL Compute Device
Work items executing an OpenCL kernel has access to four distinct memory hierarchies [12, 13].

- **Global Memory**: All the work-items in all work-groups has read and write access to this memory. Host can read and write to global memory through memory commands placed on command queue. In a GPU, Global memory is implemented as DRAM. Hence the access latency is high. In NVIDIA GPUs, peak read performance occurs when all the work items access continuous global memory locations. This is known as coalesced memory access and is shown in Figure 3-2. Non-coalesced access is shown in Figure 3-3.

- **Constant Memory**: Part of the global memory that is read only to all work-items in all work-groups. Some devices provide fast cached access to constant memory. The memory is allocated and data is copied by the host.

- **Local Memory**: The memory region local to a work-group, shared by all the work-items in that work-group. OpenCL devices like NVIDIA and ATI GPUs provide dedicated local memory on the device which is as fast as registers. In some other devices local memory is mapped to global memory. In NVIDIA GPUs, local memory is referred as shared memory. Shared memory is implemented as banks. When multiple work items (threads) in a work group (block) access the same bank, bank conflicts occur which results in the serialization of access. Figure 3-4 shows the bank conflicts in Banks 0, 2 and 5. Local memory provides very fast broadcast when all the work-items read the same location.

- **Private Memory**: The lowest level in the memory hierarchy which stores the local variables of a work-item. These are mostly hardware registers. Host cannot read and write to local or private memories. OpenCL hardware does dynamic partitioning of register file among all the work-groups that are scheduled to run in this compute unit. Once assigned to a work-group, the register is not accessible by work-items in other work-groups. If a work-group over uses the registers, other work-groups will starve. This limits the number of work-groups that can be run in parallel. Hence overuse of the local variables may result in the reduced number of work-groups simultaneously getting scheduled for processing. Moreover if work-group runs of out registers, the extra local variables are stored in global memory which slows down the execution. Local arrays in a kernel are stored in global memory resulting in more cycles for the access.

A high level view of OpenCL device architecture is shown in Figure 3-5. OpenCL provides two levels of synchronization

1. Synchronization of work-items in a work group: This is a barrier synchronization which ensures all the work-items execute the barrier before proceeding to execution beyond the barrier.
2. Synchronization between commands enqueued in a command queue: This ensures that all the commands queued before the barrier have finished execution and the resulting memory is visible to the commands after the barrier before they begin execution.
As mentioned before, K-means algorithm works in iterations. Each iteration is dependent on the results (new cluster centroids) from the previous iteration. Hence the iterations cannot be parallelized. What we can parallelize is the computations done within an iteration. Algorithm 1 shows the sequential $K$-means clustering method.

Algorithm 1 General Algorithm

1: dataPoints = read_data_points()
2: models = random(dataPoints)
3: for $i = 0$ to Number of Clustering iterations do
4:   centroidStats = compute_statistics(dataPoints, models)
5:   models = compute_new_models(centroidStats)
6: end for

The initial set of centroids is randomly chosen from the data points. The $compute\_statistics()$ function iterates over all the data points and finds the closest centroid in each model. $compute\_new\_models()$ aggregates the statistics from the $compute\_statistics()$ function to find the new models for next iteration. The $compute\_statistics()$ function is the main work horse as it does all the computation. In the thesis, we target this function for parallelization on an OpenCL device. We also implemented the kernel for the aggregation step so that we do not have to transfer data between CPU and the device across clustering iterations.

4.1 Parallelization Issues

For parallelizing an algorithm on an OpenCL device, we need to address the following issues:

4.1.1 Concurrency

The two main entities that can be partitioned in a K-means clustering problem are data points and models. Each model can be mapped to a problem instance which operates on data. OpenCL standard provides 2 levels of parallelism:
Table 4-1. Parallelization Methodologies

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Models</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Parallelism</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Data Parallelism</td>
<td>-</td>
<td>1, 2</td>
</tr>
<tr>
<td>Concatenated Parallelism</td>
<td>2</td>
<td>1, 2</td>
</tr>
</tbody>
</table>

1. Parallelism among work-groups

2. Parallelism among work-items within a work-group

As shown in Table 4-1, we devised three parallelization methodologies based on how data and models uses these two levels of parallelism. These methodologies are for the implementation of `compute_statistics()` function. `compute_new_models()` function remains the same for all the three strategies.

4.1.2 Memory Constraints

As mentioned in Chapter 3, OpenCL device provides a hierarchical memory structure. Before deciding on which hierarchy to place a data structure, we need to carefully identify the memory access pattern. The objects that are accessed over and over again will result in performance penalty if stored in global memory and the fast local memory is limited by size. The options to store the input (data records and models) are as follows:

- Keep both the models and data points in global memory.
- Keep data points in global memory. Load the models to local memory. Each data point is read from global memory to the private register and computation is performed against all the models.
- Keep models in global memory. Load data points to local memory. For each data point in local memory performs computations against the models.

We also need to keep the data structures that store the centroid statistics after each computation. The options to store the statistics data structures are

- Update the statistics in global memory.
- Update the statistics in local memory and after partial execution update the results to global memory.
4.1.3 Synchronization and Atomic Operations

Our implementation of clustering has two OpenCL kernels. Even though kernel calls are asynchronous with the host, the kernel invocations in a particular stream are serialized. Hence there is no need for explicit synchronization between the two kernel invocations. We need to identify synchronization points within a work-group and use barrier synchronization for synchronizing the work-items.

In parallel implementation of K-means algorithm, the work-items compute the distances in parallel. We have memory in-consistency because many work-items can concurrently update the statistics of centroids. We use the atomic operations provided by the OpenCL standard to achieve memory consistency. Excessive use of atomic operations can slow down the execution. Since we have two options to store the statistics data structures, we can have atomic operations in

- Global memory: This requires a large number of cycles.
- Local memory: This is faster than global memory atomic updates. However, it can result in contention when many work-items in a work-group update the same location in local memory.

4.1.4 Data Transfer Between Host and Device

Before launching a kernel, all the required data has to be copied to device global memory. After execution the results are copied back to the host. If there are multiple kernels, there can be frequent transfer of data between host and device. This reduces performance.
In this chapter, we present the design and implementation of the three parallelization strategies mentioned in Table 4-1. For efficient indexing, we store data points and models in linear arrays. The two arrays copied to the OpenCL device global memory as part of pre-processing step. To keep track of the statistics after `compute_statistics()` kernel, we keep two additional arrays in global memory. One array for keeping track the number of closest points to centroid in the models and second array for the sum of points ($\sum x, \sum y, \sum z$, etc) closest to each centroid in the models. The `compute_new_models()` kernel uses these two arrays to compute the new set of models for use in the next iteration. There is no data-transfer between the OpenCL device and host during the clustering iterations.

For each methodology, we analyze the computational and data access complexities for an iteration. We use the following conventions for our analysis. Data set has $N$ records where each record is of $D$ dimension. We choose random $M$ models, each with $K$ cluster centroids randomly chosen from $N$ records. We also discuss the Algorithm and OpenCL code for each methodology. For simplicity of description, we present the OpenCL code for two-dimensional data points. All of our kernels use 256 work-items in a work-group. This is denoted by the macro `WORK_ITEMS_PER_WORK_GROUP` which is abbreviated as `WIPWG` in the code listings.

5.1 Task Parallelism

In task parallelism, the tasks are partitioned among the compute units. Each model is mapped to a work-group. Each work-group reads the whole data-record from the device global memory. The work-group computes the closest centroid of each data point in the respective model. Figure 5-1 depicts task parallelism.

With in a work-group, the computation is partitioned among the work-items. Each work-item reads a chunk of data record and finds the closest centroid in the model. In
this way all the work-items in a work-group reads the mapped model for computation for all the data points. Reading the model every time from global memory affects performance. This makes model an ideal candidate for keeping in local memory. At the start of kernel, the work-items load the models to local memory. All the work-items have to wait till the model is completely loaded in local memory. This requires the use of a a barrier synchronization.

Each work-group executes the algorithm depicted in Algorithm 2.

**Algorithm 2 Task Parallelism**

1: Load the model \(m\) corresponding to this work-group to local memory
2: barrier synchronize()
3: for each data point \(p\) assigned to a work-item do
4: statistics = find closest centroid\((p, m)\)
5: atom_update(statistics) in local memory
6: end for
7: barrier synchronize()
8: write back statistics from local to global

We now discuss in detail the OpenCL implementation of Algorithm 2 for two dimensional records. The work-items in a work-group loads the points in a particular model from global memory to local memory. The work-group barrier synchronization function ensures that the required model is loaded before proceeding to computation phase. We keep two additional linear arrays \((sCount\) and \(sSum)\) in local memory to store the computational result. \(sCount[i]\) stores the number of data points closest to the \(i^{th}\)
centroid in the model. $sSum[i]$ stores the cumulative sum ($\sum x, \sum y$ for two dimensions) of each data point closest to the $i^{th}$ centroid in the model. Listing 5-1 shows the distance computation phase of the kernel.

```c
// dData in Global Memory
// sModels, sCount, sSum in Local Memory
// work-item id
size_t tx = get_local_id(0);
unsigned pointsPerWorkItem = NO_DATA / TPWG
for (unsigned t = 0; t < pointsPerWorkItem; ++t)
{
    unsigned dataIndex = t * WIPWG + tx;
    // Coalesced access from global memory
    Point point = dData[dataIndex];
    int minDist = MAX_DISTANCE;
    int minIndex = 0;
    for (unsigned j = 0; j < MODEL_SIZE; ++j)
    {
        Point c = sModels[j];
        int dist = computeDistance(point, c);
        minIndex = min(dist, minIndex);
    }
    atom.add(sCount + minIndex, 1);
    atom.add(&(sSum[minIndex].x), point.x);
    atom.add(&(sSum[minIndex].y), point.y);
}
barrier(CLK_LOCAL_MEM_FENCE);
```

Listing 5-1. Computation phase of task parallelism

Each work-item in a work-group reads a portion of data from $dData$ array stored in device global memory. The data read uses the coalesced pattern mentioned in Chapter 3. In each iteration of the inner loop, a data point is compared against the model stored in local memory array $sModels$. At each time-frame all the work-items in a work-group reads from the same location of $sModels$ array in local memory. This makes the hardware broadcast the data read to all the work-items in one shot. Once the closest point is identified, the sum and the count is updated (atomic) in the local memory. Atomic update is required because many threads can update $sCount$ and $sSum$ arrays corresponding to the same centroid in a model. The barrier synchronization step
ensures all the computation with in this work-group is completed before we proceed to
the write back stage.

```c
unsigned index = bx * MODEL_SIZE + tx;
if (tx < MODEL_SIZE)
{
    dCount[index] = sCount[tx];
    dSum[index] = sSum[tx];
}
```

Listing 5-2. Write back phase of task parallelism

Listing 5-2 shows the data write back phase. The sum and the count obtained from the
computation phase is updated from local memory to global memory. Since each block
is mapped to a work-group, there are no conflicts between the blocks. Hence global
memory update does not need atomic functions.

In task parallelism, data is read $M$ times from global memory. This access
contributes to the majority of run time. If $M$ is less than the number of computing units
in the hardware, then we will not be using the hardware efficiently. The local memory
conflicts will be large if $K$ is small as all work-items in a work-group will be trying to
update one of the centroids in the model. The total execution time is calculated as the
sum of data access time and time for computation. Analysis: For each model, data is
read once. There are $M$ such models.

\[ Data\ \text{accesses} = O(MN) \]
\[ Computations = O(MNKd) \]

5.2 Data Parallelism

In data parallelism, data is partitioned across work groups and work-items with in a
work-group. All work-groups work on the same model. i.e. each invocation of OpenCL
kernel will be working on one model and the whole dataset. Figure 5-2 depicts data
parallelism. With in a work-group, each work-item reads a sub-chunk from the chunk of
data assigned to this work-group and computes the closest centroid in the model. As
in task parallelism, the model array is accessed by all work-items, hence stored in local memory. All the work-groups in a kernel loads the same model to local memory. The barrier synchronization step remains the same as in task parallelism. Each work-group executes the algorithm depicted in Algorithm 3.

**Algorithm 3** Data Parallelism

```plaintext
1: for every model \( m \) do
2: Load the model \( m \) to local memory
3: barrier_synchronize()
4: for each data point \( p \) assigned to a work-item do
5: statistics = find_closest_centroid(p, m)
6: atom_update(statistics) in local memory
7: end for
8: barrier_synchronize()
9: atom_update(statistics) from local to global
10: end for
```

The OpenCL implementation of Algorithm 3 is as follows. Listing 5-3 shows the computational phase. Each work-group reads chunk of the data array. The chunk is split among the work-items in the work-group. Work-items access the data array in coalesced pattern. The number of sub-chunks read by a work-item is controlled by the macro `POINTS_PER_WORK_ITEM` which is abbreviated as `PPWI` in the code listing. `PPWI` controls the number of work-groups. When `PPWI` is more, the number of work-groups...
is less and vice versa. The local memory access broadcast, atomic updation of \(sSum\) and \(sCount\) arrays remains the same as in the computation phase of task parallelism (Listing 5-1).

```
// dData in global memory
// work-group id
size_t bx = get_group_id(0);
// work-item id
size_t tx = get_local_id(0);
unsigned bIndex = WIPWG * PPWI * bx + tx;
for (unsigned p = 0; p < PPWI; ++p)
{
    unsigned dataIndex = bIndex + (p * WIPWG);
    Point point = dData[dataIndex];
    // Same code from task parallelism follows here
}
barrier(CLK_LOCAL_MEM_FENCE);
```

Listing 5-3. Computation phase of data parallelism

The write back phase of data parallelism is different from task parallelism. As shown in Listing 5-4, write back phase involves atomic updates in global memory. As all the work-groups in a kernel work on the same model, they update the same location of statistics arrays in the global memory.

```
if (tx < SET_SIZE)
{
    atom_add(dCount + cIndex, sCount[tx]);
    atom_add(&(dSum[cIndex].x), sSum[tx].x);
    atom_add(&(dSum[cIndex].y), sSum[tx].y);
}
```

Listing 5-4. Write back phase of data parallelism

The data access and computational complexities remains the same as in task parallelism. Multiple kernel invocations and atomic updates in global memory contribute to an additional overhead in data parallelism.

### 5.3 Concatenated Parallelism

Both task parallelism and data parallelism require accessing data multiple times in a single iteration. We devised a new methodology, called concatenated parallelism,
Figure 5-3. Concatenated Parallelism

with the primary aim of reducing this data access. In concatenated parallelism, data is partitioned across work-groups and work-items and computation is partitioned across work-items in a work-group. All the work-groups work on all the models. Concatenated parallelism is shown in Figure 5-3. Like data parallelism, each work-item with in a work-group reads a sub-chunk of data assigned to this work-group. Each data-point from the sub-chunk is compared against all the models and the closest centroid in each model is identified. Thus each work-item in a work-group accesses all the models for each data point. Keeping all the models in local memory ensures fast access by all the threads. Each work-group executes the algorithm shown in Algorithm 4.

Algorithm 4 Concatenated Parallelism

1: Load all the models to local memory
2: barrier_syncronize()
3: for each data point $p$ assigned to a work-item do
4:   for each model $m$ in local memory do
5:     statistics = find_closest_centroid($p$, $m$)
6:     atom_update(statistics) in local memory
7:   end for
8: end for
9: barrier_syncronize()
10: atom_update(statistics) from local to global

Listing 5-5 shows the OpenCL implementation of computational phase of concatenated parallelism. $sCount$ and $sSum$ arrays now store the count and sum respectively for all
the models. Like data parallelism the work-items read sub-chunks of data in coalesced pattern. The local memory stores all the models. The local memory access pattern as in task and data parallelism result in each work-item starting computation with the centroids of first model. This results in a fast broadcast during the read. But during the atomic updation of sCount and sSum all the work-items update the locations corresponding to the same model. This results in high contention causing serialized access which in turn degrades performance. High contention atomic update is shown in in Figure 5-4. The are four work-items (W0, W1, W2, W3, W4) in the work-group update the same model in a particular time frame.

We devised a reduced contention methodology using distributed access pattern. In distributed access pattern, the $i^{th}$ work-item in a work-group starts computation with the model having index $i \mod \text{NO\_MODELS}$. In this way, $i^{th}$ work-item will start updating the locations corresponding to model $i \mod \text{NO\_MODELS}$ in the sCount and sSum arrays. This reduces the contention due to conflicting access. One drawback of this method is
that we cannot utilize the broadcast feature provided by the local memory hardware.

Our experiments show that contention can be a major bottleneck compared to the performance improvement from broadcast. Reduced contention atomic update is shown in Figure 5-5. Work-items access different models in a time frame.

```c
// dData in global memory
// sModels, sCount, sSum in shared memory
// work-group id
size_t bx = get_group_id(0);
// work-item id
size_t tx = get_local_id(0);
unsigned bIndex = TPWG * PPWI * bx + tx;
for (unsigned p = 0; p < PPWI; ++p)
{
    unsigned dataIndex = bIndex + (p * TPB);
    // coalesced read
    Point point = dData[dataIndex];
    for (unsigned i = 0; i < NO_MODELS; ++i)
    {
        int minDist = MAXDISTANCE;
        int minIndex = 0;
        // Distributed model access
        unsigned gSetIndex = ((i + tx) % NO_MODELS) * MODEL_SIZE;
        for (unsigned j = 0; j < MODEL_SIZE; ++j)
        {
            unsigned setIndex = gSetIndex + j;
            Point c = sModels[setIndex];
            int dist = computeDistance(point, c);
            minIndex = min(dist, minIndex);
        }
        atom_add(sCount + minIndex, 1);
        atom_add(&sSum[minIndex].x, point.x);
        atom_add(&sSum[minIndex].y, point.y);
    }
}
barrier(CLK_LOCAL_MEM_FENCE);
```

Listing 5-5. Computation phase of concatenated parallelism

As shown in Listing 5-6, the write back phase of concatenated parallelism remains similar to data parallelism with each work-group doing an atomic write back of all the model statistics to global memory.
for (unsigned i = 0; i < centPerThread; ++i)
{
    const unsigned cIndex = tx * centPerThread + i;
    if (cIndex >= NO_CENT) break;
    atom_add(dCount + cIndex, sCount[cIndex]);
    atom_add(&(dSum[cIndex].x), sSum[cIndex].x);
    atom_add(&(dSum[cIndex].y), sSum[cIndex].y);
}

Listing 5-6. Write back phase of concatenated parallelism

One of the major improvement in concatenated parallelism is the number of times data is read. Data is read only once in each clustering iteration. Data access complexity is $O(N)$ unlike the data and task parallelism where the complexity is $O(MN)$. The computations remains the same as in task and data parallelism. Concatenated parallelism does not involve the multiple kernel invocation overhead as in data parallelism. The overhead due to atomic updates in global memory is much less when compare to the time spent in reading data multiple times as in task parallelism.
CHAPTER 6
PARALLEL $k$-MEANS ALGORITHM USING KD-TREE

The algorithm described in Chapter 4 requires the distance computation with each data point for all the models. As introduced in Section 2.3, we use a KD-tree to reduce this computation by ordering the data points. A KD tree is created from data points. The creation of KD tree for multi dimensional data is described in detail in [1]. The KD tree creation is done only once in the whole clustering process. Once the KD-tree is created, the centroids in each model traverse the KD-tree and gets mapped to data points in the lead nodes. Algorithm 5 shows the modified version of Algorithm 1 and uses KD-tree for partitioning the data points. Figure 6-1 shows the KD-tree with data points mapped centroids from models. The creation and traversal of KD-tree is performed by the functions `create_kd_tree()` and `create_leaf_node_to_centroid_mapping()` respectively. These two steps are done in CPU because the irregular structure of KD-tree is not well suited for the SIMT architecture of the OpenCL device. Once the KD-tree is created and mapping of centroids to data points are created, the mapping information is copied to the device. After computation, the new centroids are copied to the host machine to create the mapping information for next iteration. The copying of mapping information to the device and the new centroids from device to host is performed for each clustering iteration and creates additional overhead.

![KD-tree with Data Points and Mapped Centroids on Leaf Nodes](image)

Figure 6-1. KD-tree with Data Points and Mapped Centroids on Leaf Nodes
Algorithm 5 Pruning using KD-tree

1: dataPoints = read_data_points()
2: models = random(dataPoints)
3: kd_tree = create_kd_tree(dataPoints)
4: for $i = 0$ to Number of Clustering iterations do
5:  mapping = create_leaf_node_to_centroid_mapping (kd_tree, models)
6:  copy_to_device(mapping)
7:  centroidStats = compute_statistics(mapping)
8:  models = compute_new_models(centroidStats)
9:  copy_from_device(models)
10: end for

Similar to the parallel implementations described in Chapter 5, we can have task, data and concatenated parallelism for the KD-tree based implementation as well.

1. **Task Parallelism:** Each device work-group performs computations for a model. The work-items in a work-group read the data points and computation is performed for the mapped centroids of this model. The centroids in different models get mapped to same data points. Hence each work-group must read the entire data.

2. **Data Parallelism:** Each kernel invocation performs computation for a model. The data points are partitioned among the work-groups. i.e. each leaf node of a KD-tree gets mapped to a work-group. The work-items in a work group reads the data points and computation is performed for the mapped centroids of the model. As in task parallelism, data is read multiple times.

3. **Concatenated Parallelism:** As in data parallelism, each leaf-node of KD-tree is mapped to a work-group. The work-items in a work-group performs reads the data points and performs computations for the mapped centroids of all the models. In this approach, the data will be read only once and there is no overhead of multiple kernel calls for each model as in data parallelism. Let $\beta$ be the pruning factor which is the fraction by which the computation is reduced after pruning by traversing the KD-tree. We implemented three different approaches for concatenated parallelism which is discussed in Section 5.3

\[
\begin{align*}
\text{Data access time} & = O(N) \\
\text{Computation time} & = O\left(\frac{MKNd}{\beta}\right)
\end{align*}
\]
6.1 Concatenated Parallelism Using KD-tree

In this section we discuss three different implementations of concatenated parallelism using a KD-tree. Each of these implementations differs in the local/global memory access patterns.

6.1.1 Direct Access

Each leaf node is mapped to a device work-group. The work-group performs computations for all the data points in this leaf node and the corresponding set of mapped centroids from the models. Figure 6-2 shows the mapping from leaf nodes to work items.

The centroid data structures (sum and the count arrays) are stored in the local memory. Likewise normal $K$ means, we use the atomic update function provided by device hardware to achieve memory consistency when work-items in a work-group update the same location. If the number of centroids mapped to a particular leaf node of KD-tree less, all the work-items can update (atomic) the same locations of the sum and the count arrays in the local memory. This increases the contention and results in serialized access degrading the performance. The next two approaches describes methods to reduce the contention.
6.1.2 Distributed Access

In this approach, we reduce the local memory update contention by using a distributed access pattern. Instead of mapping a leaf node to a work-group, the work-group spans across many leaf nodes. In this way, the work-items in a work-group update different locations in the local memory. This reduces the contention in updating the local memory. Figure 6-3 shows the mapping from leaf nodes to work items.

One drawback of this approach is that the number of global memory access for data increases. This is because computation in a leaf node is performed by multiple work-groups.

6.1.3 Eliminating Memory Conflicts Using Replication

In this approach, we replicate the data structures for each work-group and work-item in the device global memory. Like direct mapped approach, each leaf in the KD-tree is mapped to a work-group. Each work-item operates on its own memory location so that there won’t be any global memory conflicts. After computation, another device kernel aggregates the results from all the replicated models. One drawback with this approach is that replication results in an increase in the memory footprint.
CHAPTER 7
EMPIRICAL EVALUATION

We analyze the three clustering approaches Task Parallelism (TP), Data Parallelism (DP), and Concatenated Parallelism. We determine the best parallelization strategy on multi-core hardware. We also study the performance impact of using index based data structures.

7.1 Datasets

We used randomly generated data set of size 1 Million points with each data point of varying dimensions. The initial centroids for the models are chosen randomly from the data set. We fix the number of cluster centroids to be 10 the number of clustering iterations is fixed to 10.

7.2 Benchmark Machines

We conducted our experiments on three different hardware configurations. All the machines run on 64 bit Linux operating system.

1. The machine with an NVIDIA FERMI Generation GeForce GTX 480 GPU clocked at 1.4 GHz. The machine has a four core Intel Xeon Processor clocked at 2.8 GHz. The GPU has 1.6 GB global memory and 48 KB local memory. It has 15 compute units and each compute unit has 32 processing elements resulting in a total of 480 streaming processor cores.

2. The machine with an ATI FirePro V7800 GPU. The machine has a four core Intel Xeon Processor clocked at 2.8 GHz. The GPU has 1 GB global memory and 32 KB local memory. It has 18 compute units. Each compute unit has 80 processing elements resulting in a total of 1440 streaming processor cores.

3. The machine with 8 quad core AMD Opteron processors clocked at 3.6 GHz.

7.3 Comparison of Three Clustering Approaches on a 32-core CPU

Figure 7-1 shows the comparison of TP, DP and CP on 32 core CPU machine. We used two dimensional data and study the results by varying the number of models. The graph clearly shows CP gives much better performance compared to DP and TP. DP is faster than TP initially because in TP when the models are less, the number of work-groups are less. Hence the all the compute units of hardware is not used.
effectively. As the number of models increase, the number of work-groups increase utilizing all the compute units of hardware. In DP as the number of models increase, so does the number of kernel calls. This causes additional overhead that reduces the performance.

### 7.4 Comparison of Three Clustering Approaches on FERMI GPU

Figure 7-2 shows the comparison of TP, DP and CP on NVIDIA FERMI GPU. The data set and models configuration remains the same as in 7.3. The results are similar to 32 core CPU with CP giving the best performance and TP performing better than DP as the number of models increase.

### 7.5 Performance Comparison of Different Hardware

From sections 7.3 and 7.4, we conclude that concatenated parallelism gives the best performance on a multi core hardware. Now we determine which of the three hardwares gives the best performance for concatenated parallelism. Figure 7-3 shows the comparison of CP on 32 core CPU, NVIDIA FERMI GPU and ATI GPU. GPUs provide close to $10^X$ performance improvement over the 32-core CPU. ATI and FERMI GPU gives comparable performance. Even though ATI has more execution cores,
FERMI has a wider memory bus. In rest of the experiments we use the FERMI GPU as the OpenCL hardware.

### 7.6 Comparison of the Three KD-tree Algorithms

Figure 7-4 shows the comparison of the three KD-tree implementations of concatenated parallelism described in section 6.1. Distributed mapped implementation gives the best performance due to reduced contention. The implementation with no
conflicts is slower than distributed method due to more number of global memory accesses.

7.7 KD-tree vs Direct Implementation of Concatenated Parallelism

In this section, we compare the best KD-tree implementation with the direct implementation of concatenated parallelism. From section 7.6, we conclude that direct mapped implementation of KD-tree gives the best performance. As shown in Figure 7-5, the KD-tree implementation gives an average of $2X$ performance improvement over the normal implementation.

7.8 KD-tree vs Direct Implementation for Varying Dimensions

In this section, we compare the performance of KD-tree and direct implementation when the dimensionality of data changes. The number of models is fixed as 10. Results are shown in Figure 7-6, the performance of KD-tree implementation decreases as dimensionality of data increases. This is because as the dimensionality of data increases the pruning obtained from KD-tree decreases. Moreover there is additional overhead from KD-tree traversal, data transfer between the host and the device.
Figure 7-5. KD-tree vs Direct Implementation of Concatenated Parallelism

Figure 7-6. KD-tree vs Direct Implementation for Varying Dimensions

7.9 Discussion

We conclude that concatenated parallelism gives better performance for data intensive applications. Our results show that memory access patterns is one of the key factors that determines the overall performance of any algorithm running on many core hardware. Although our description assumes that all the models have equal number of
centroids. But with a small change in the algorithm we will be able to support models with different number of centroids as well.
CHAPTER 8
CONCLUSION

As the processors are evolving, the memory traffic is becoming a major bottleneck. In the thesis, we presented an approach for reducing the memory traffic for ensemble clustering. Our experimental results clearly show that reducing memory traffic can lead to significant performance gains. The additional advantages of using complex data structures to reduce the total amount of computation has benefits but the overall reduction is time is much smaller compared to the advantages of concatenated parallelism due to substantial reduction in the traffic. We believe that the concatenated parallelism approach presented in this paper is relatively general can be used to develop a framework for automatic parallelization of ensemble computing applications.
REFERENCES


BIOGRAPHICAL SKETCH

Girish Ravunni Kutty received his Bachelor in Computer Engineering from Department of Computer Science and Engineering, National Institute of Technology, Calicut, India in 2005. After graduation, he worked in software industry for four years. He graduated with his Master of Science in computer engineering from the Department of Computer & Information Science and Engineering, University of Florida, Gainesville in 2011. He served as a Research assistant in the department during his master’s program. His research interests include High performance computing and Auto tuning for heterogeneous systems.