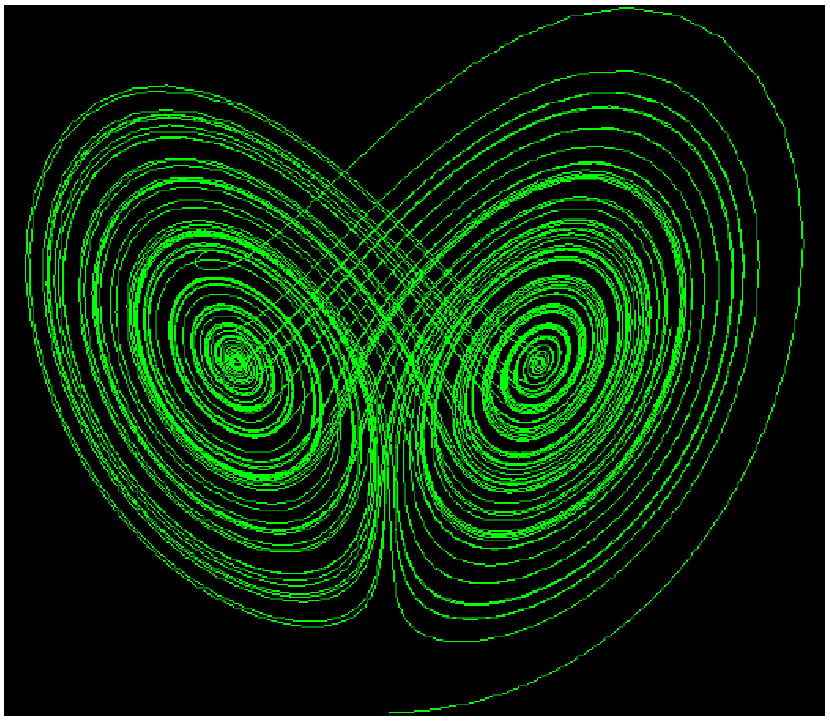


GPU Accelerated Computing with Data Mining Algorithms for Advanced Analytics

Jeremy Johnathan Williams

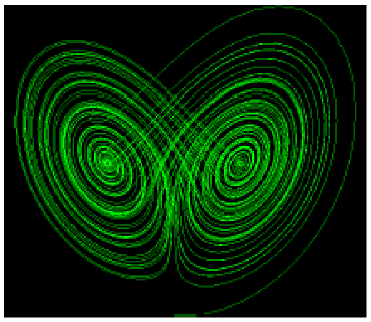
Departments of Mathematics, Physics and DACSO

MASTERS THESIS

Modelling for Science and Engineering

GPU Accelerated Computing with Data Mining Algorithms for Advanced Analytics

Jeremy Johnathan Williams



July 2019

Universitat Autònoma de Barcelona

ABSTRACT

Advanced Analytics success requires both a vigilant deep business approach and technical knowledge. Uniting both, helps to generate performance improvements and accelerate computation in applications for unknown insight.

The research will focus on the acceleration of a typical advanced analytics process using GPU accelerated computing. The central algorithm of the data mining process will be the
k-means clustering algorithm, which is a classical algorithm for unsupervised machine learning.

The research will be an experimental study with a motivation to successfully design, optimize and evaluate a GPU-accelerated implementation of the k-means clustering algorithm.
The input data will be an official real dataset that was extracted using a Data Extraction System and will fit into the GPU memory. Performance engineering methodologies will be used to develop the optimized version. The improved version will be executed on a GPU-accelerated computer system with OpenACC directives. The OpenACC directives will target both the CPU and GPU architectures and launch computational code on them to easily accelerate the performance of the advanced analytics process.

The scalability of the algorithm using Multi-Core CPUs to solve the problem and combine computation between CPU and GPU in a heterogeneous system was investigated.

The Performance Analysis process of multiple ‘K’ clusters and ‘D’ multidimensional experimentations along with application insight were conducted. Data visualization results of the GPU-accelerated computer system discovered were also deliberated.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis advisor, Dr. Juan Carlos Moure Lopez for the useful comments, remarks and engagement through the learning process of this master thesis. I would like to thank the director and the selection committee of Department of Computer Architecture and Operating Systems (CAOS) organized by the School of Engineering, UAB for allowing me the opportunity to develop my research training in the area of Architecture and Computer Technology, in particular, Structure of Computers, Computer Architecture, Operating Systems, Distributed, Parallel and High Performance Processing and its applications leaded by my supervisor at Dr. Juan Carlos Moure Lopez. Furthermore, I would like to thank all of the MMSE Professors, Committee Members and Coordinator, Professor Anna Cima, for helping me with the selection of my training, educational development, progression of my topic, research experience, thesis structure, class accommodations in the Department of Computer Architecture and Operating Systems, Department of Physics and Department of Mathematics, in particular, Centre de Recerca Matemática. Also, I would like to thank my loved ones, who have supported me throughout entire process, both by keeping me harmonious and helping me put the pieces of this master thesis together. I will be forever grateful to you all.

CONTENTS

1. INTRODUCTION

SINCE the beginning of time, business and technology have been disengaged as two completely different regions of capable undertakings and futuristic developments. These anomalies inspired essential improvements in industry, research and extraordinary findings over the last few centuries; towards what we know today as the 21st century.

To improve today's unforeseen findings, there are many tasks, technologies, applications, techniques and skills that need to be investigated, explored and explained from the preliminary understanding; which is commonly known as "Analytics". Fundamentally, business users and data specialists, perceive "Analytics" as the detection, clarification, explanation, arrangement, visualization and other meaningful information grouped from the analysis of designated data sources; to make effective proficient resolutions to improve the needs within an organization.

Analytics comprises of two major areas: Business Intelligence and Advanced Analytics. Business Intelligence (BI) customarily concentrates on the effective application of various metrics to direct planning and quantify past enactment of centralized intelligence of an organization. BI consists of reporting, OLAP (online analytical processing) and data extractions/querying to solve questions like "How often", "How many" or "What exactly happened". On the contrast, Advanced Analytics (AA) goes further and deeper than BI, with the usage of more complex and technical methods to make forecasts or similarities of groups, of a designated data source, that were otherwise unable to be detected by normal systematic procedures of BI. AA solves exclusive questions like "What`s Next?", "How effective?", "What will happen if this is changed?", "What improvements/enhancements can be developed?", "What are the likenesses/similarities?" or "What can we understand from this formless information?".

These two areas have accumulated, in recent years, the formation of the development and enhancing authority of unidentified understandings. In this changing world and development of large amounts of data, AA success requires both a vigilant deep business approach and technical knowledge. Uniting both, helps to generate performance improvements and accelerate computation in applications for unknown insight.

2. OBJECTIVES

The research will begin with the characterizations of Accelerators, GPU Computing and Data Mining. The focus will be on the acceleration of a typical advanced analytics process using GPU accelerated computing. The central algorithm of the data mining process will be the k-means clustering algorithm, which is a classical algorithm for unsupervised machine learning.

Firstly, the research will be an experimental study with a motivation to successfully design, optimize and evaluate a GPU-accelerated implementation of the k-means clustering algorithm. The input data will be an official real dataset that was extracted using a Data Extraction System and will fit into the GPU memory.

Secondly, performance engineering methodologies will be used to develop an optimized version. Then the improved version will be executed on a GPU-accelerated computer system with OpenACC directives. The OpenACC directives will target both the CPU and GPU architectures and launch computational code on them to easily accelerate the performance of the advanced analytics process. Thus, performing and accomplishing a GPU-accelerated application together with of the k-means clustering algorithm.

Finally, the scalability of the algorithm using Multi-Core CPUs to solve the problem and combine computation between CPU and GPU in a heterogeneous system will be investigated. Then a performance analysis process of multiple ‘K’ clusters and ‘D’ multidimensional experimentations will be conducted; with data visualization results of the GPU-accelerated computer system discovered. Thus, ultimately achieving an improved GPU-accelerated implementation; to effectively provide advanced analytics insight.

To explicate the acceleration of a typical advanced analytics process using GPU accelerated computing, we must first explain Accelerators, GPU Computing and Data Mining.

3. LITERATURE REVIEW

To properly use Accelerated Computing via GPUs with Data Mining Algorithms for Advanced Analytics, it is important to declare, investigate and understand all essential definitions and terminologies.

3.1 Accelerators

Today, many scientists and researchers are developing ways to help complex computing applications to reduce performance bottlenecks, decrease overloaded CPU usage, improve memory access, better network connections, recover software processing power and diminish data storage capacity usage on slower devices. Depending on the computing problem and/or task at hand, this can be achieved by accelerators to improve overall performance.

Newton's Second Law, states how an object's acceleration is result of the amount of force applied to it with respect to time. And that if we intensify the force on this object, the acceleration of this object will also strengthen **[7].** But, how does this apply to algorithmic applications? "Accelerators" such as GPUs or other multicore vector coprocessors are complementary for improving algorithms execution time. These accelerators have memory bandwidths greater than CPUs that are enhanced for parallel throughput rather than latency of single threaded performance **[8]**.

According to Figure 1, CPUs and GPUs (Accelerators) are designed and operate very differently. CPUs have a latency oriented system to achieve three (3) purposes,

1) **Have powerful Arithmetic Logic Units (ALUs)** (to execute arithmetic and logic operations for reduced operation latency)
2) **Have large caches** (to transform long latency memory accesses to short latency cache accesses)
3) **Have a sophisticated control structure** for branch prediction and speculative fetching (to avoid delays on data dependencies executions).

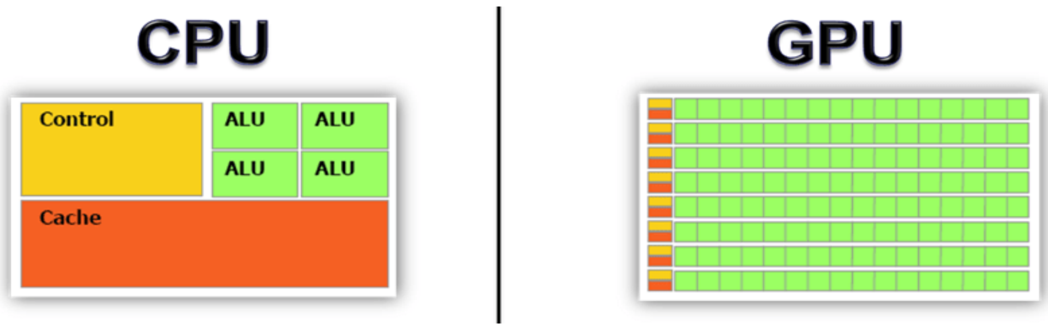


Figure 1: CPU vs GPU oriented system description including data transfer and pipelining **[5, 16]**

In contrasts, GPUs have a throughput oriented system to achieve four (4) purposes,

1) **Have very small caches** (to boost memory throughput),
2) **Have a basic simple control structure** (with no branch prediction and data forwarding necessary),
3) **Have energy efficient and resourceful ALUs** (to execute many arithmetic and logic operations latency but impressively pipelined for high throughput)
4) **Have massive number of threads to tolerate latencies** (via thread logic for massive parallelism to reduce and/or remove latency/inactivity of an executed computational application).

3.2 GPU Computing

Historically, GPUs were used around 1980s, then became popular by Nvidia in 1999 with the GeForce 256 known as the first GPU in the world **[13, 14]**. Today, GPUs are classified as specialized electronic circuit created to quickly control and change memory usage, with a computational application, to accelerate the creation of images for a display device. They are used in embedded systems, mobile phones, personal computers, workstations, game consoles and heterogeneous system architectures **[15].** Because of this, NVIDIA Corporation have invested and successfully developed accelerated computing applications that focuses mainly on three (3) types of solutions:

1) **Specialized Accelerated Libraries**, which are limited by what libraries are available. For instances, Domain-specific, Visual Processing, Linear Algebra, Math Algorithms etc.
2) **Programming Language Additions**, which are used for maximum parrellel flexibility, which in most cases difficult to learn and time consuming to implement. For instances, C, C++, CUDA, FORTRAN, Python etc.
3) **Compiler Directives**, which are easily used to accelerate applications. For instances, OpenACC via GPU Directives that are simple to use, open and powerful for massive parallel power on the GPU **[5].**

And according scientists and researchers, OpenACC Directives are in high regard as a favorable GPU computing approachable to providing faster application performance and significantly reduced programming effort to existing code **[26, 27].**

Focusing on the computing power of the GPU, computation and the usage of GPUs for parallelization has become most significant to researchers and scientists; where the general-purpose scientific and engineering computing has been attained. This achievement is known as “GPU computing”.

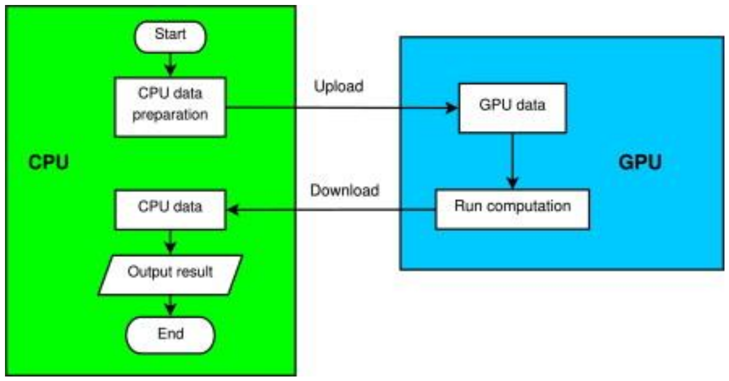


Figure 2: A simple GPU computing procedure **[5, 17]**.

From NVIDIA Corporation and Figure 2, during the GPU computing process, with a selected dataset, such a program normally operates within seven (7) phases depending on the programming application structure; with data transfer via copying being asynchronous and unified memory managed.
They are as followed,

1) **Data inputs** begins on the CPU-available memory (Host),
2) **Memory** are allocated for data outputs on the Host,
3) **Memory** are allocated for data inputs on the GPU,
4) **Memory** are allocated for data outputs on the GPU,
5) **Data inputs** are copied from Host to GPU,
6) **GPU kernel or routine** compiled is activate (tasks that executes on GPU, as required)
7) **Data output** are copied from GPU to Host.

Basically, a GPU computing application typically begins from the host with the required data prepared then transferred to the GPU to begin the execution. After the computational expensive job is completed on the GPU, the results are transferred back to the CPU, as required **[17, 22]**.

3.3 Data Mining

As we have discussed earlier, business users and data specialists are trying to make effective proficient resolutions to improve their access to insights and knowledge. However, most of them do not fully understand the complete process of analyzing data from different sources and then summarizing it for their unique need towards relevant information. This process is called data mining and sometimes is known as data or knowledge discovery. Technically speaking, data mining is the procedure of justifying relationships or arrangements among thousands of data tables or fields in enormous multidimensional relational databases and/or datasets **[11]**.

Data mining, in itself, is part of the process of knowledge discovery. It is used to extract understanding, from a dataset, and then converted into a humanized configuration for future use **[9, 10].**

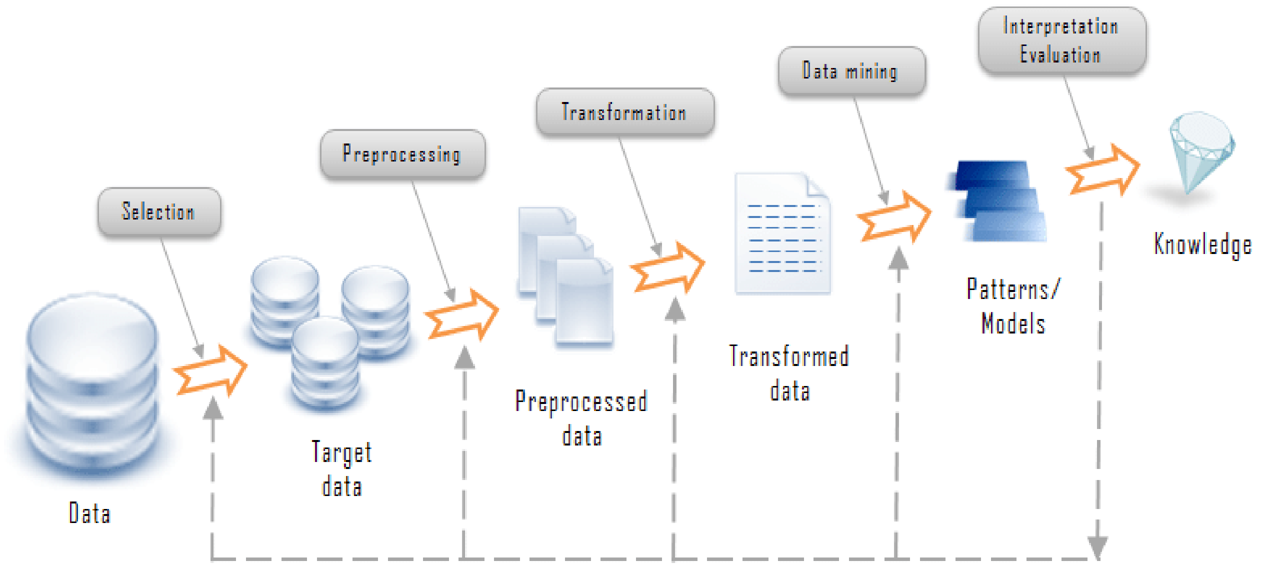


Figure 3: Data mining as a step in the process of knowledge discovery. **[9, 10]**

As we can see from Figure 3, there is a direct path to achieve knowledge; where many academic books and researcher called this "The Data Mining Process/Steps".

These references indicate four stages of data mining:

1) **Data Sources** (focusing on the problem definition),
2) **Data Exploration/Gathering** (where sampling and transformation of data are conducted),
3) **Data Modeling** (where the creation of models, testing and evaluation is executed)
4) **Model Deployment** (the action based stage, where the implementations from the results of the models are used or applied).

However, within these steps, most experts that attempt to improve the performance of the complete procedure of knowledge discovery tend to focus on the data mining stage. Why? For the reason that it is the essential part of the process where algorithmic/intelligent methods are used to extract data patterns towards and/or before knowledge is attained **[9, 10]**.

When it comes to working with big data and/or large multidimensional datasets, data mining plays an extraordinary part in the process of knowledge discovery with machine learning (ML). Concisely speaking, ML is an establish a form of learning that connects the capabilities of a computer system to make it perform learning tasks and make coherent results according to previously observed conditions and previous actions or responses; acting according to an immovable strategy. In the realm of ML and computing, it is very imperative to perform a specific task effectively without using explicit instructions via application learning complexity, a relevant type of machine learning algorithm must be selected.

Generally, there are three (3) types of machine learning algorithms. They are as followed:

1) **Supervised Learning** (consisting of a dependent variable that is to be predicted from a given set of independent variables; illustrations of this are Regression, Decision Tree, Random Forest, KNN, Logistic Regression etc.),
2) **Unsupervised Learning** (discovering unknown patterns in a dataset without initial labels **-** like clustering analysis and segmentation; illustrations of this are Apriori algorithm, K-means algorithm, Mean-Shift, DBSCAN, Expectation–Maximization, Hierarchical Clustering etc.),
3) **Reinforcement Learning** (where data is trained to make specific decisions when a machine is exposed to an environment with continuous training using trial and error; illustrations of this are Markov Decision Process, Q-Learning, State-Action-Reward-State-Action, Deep Q Network, Deep Deterministic Policy Gradient, Temporal Difference etc.) **[6].**

Additionally, this portion of the process of knowledge discovery and ML partakes countless data mining techniques with its algorithms that are used before the action-based stage.

The three (3) foremost data mining techniques discovered by researchers are as followed:

1) **Classification** (used to predict/classify each element of a dataset/database into an assigned/predefined set of classes or groups; illustrations of its algorithms are c4.5, support vector machine etc.),
2) **Clustering** (used to exam one or more classes that can be grouped/clustered together to form similarities and/or correlating results; illustrations of its algorithms are Apriori, K-means etc.)
3) **Regression** (used to estimate relationships among variables based on training processes by comparing predicted results; illustrations of its algorithms are Logistic, Lasso, support vector machine etc.) **[12]**.

4. ALGORITHM

As discussed earlier, the process of knowledge discovery and ML partakes countless data mining techniques discovered by researchers. We will now declare, investigate and understand the algorithm and distance formula used in this research.

Unsupervised learning - data mining techniques, due to continued technological advancements, have become greatly significance to academia and the industry; as large datasets (Di), and databases continue to grow in magnitude. A very popular type is called the K-means algorithm that is known to be based on “A Centroid-Based Partitioning Technique” which “...uses the centroid of a cluster, Ci, to represent that cluster. Conceptually, the centroid of a cluster is its center point. The centroid can be defined in various ways such as by the mean or medoids of the objects (or points) assigned to the cluster…“How does the k-means algorithm work?” The k-means algorithm defines the centroid of a cluster as the mean value of the points within the cluster. It proceeds as follows. First, it randomly selects k of the objects in Di, each of which initially represents a cluster mean or center. For each of the remaining objects, an object is assigned to the cluster to which it is the most similar, based on the … distance between the object and the cluster mean. The k-means algorithm then iteratively improves the within-cluster variation. For each cluster, it computes the new mean using the objects assigned to the cluster in the previous iteration. All the objects are then reassigned using the updated means as the new cluster centers. The iterations continue until the assignment is stable, that is, the clusters formed in the current round are the same as those formed in the previous round.” **[20].**

The distance formula plays a significance part in the data mining clustering process. Focusing on two points A and B, there exists four (4) types of distance formulas:

1) **Euclidean Distance (ED)** that calculates the square root of the square differences between data points of entities with formula $ED = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2}$,
2) **Manhattan Distance (ManD)** that calulate the absolute value of the differences between data points of entities, with the formula, $ManD = |A_i - B_i|$,
3) **Chebychev Distance (CD)** that calulate the absolute maximum value of the differences between data points of entities with formula, $CD = Max_i|A_i - B_i|$,
4) **Minkowski Distance (MinD)** that is a generalized formula of the ED, CD and ManD, with the formula, $MinD = \left(\sum_{i=1}^{n}|A_i - B_i|^{\frac{1}{p}}\right)^{p}$ where if p = 2 then MinD = ED, p = 1 then MinD = ManD and p = ∞ then MinD = CD **[21].**

Additionally, it has been identified that the k-means clustering algorithm has a problem of converging to a local minimum of the criterion function depending upon the selection of initial centroids. That is why selecting the proper initial centroid is very important. Also, studies have mentioned that, the k-means clustering algorithm expects the total number of desired clusters to be formed and the total number of iterations to be performed as inputs. So, predicting "k" clusters and iterations can be a problem with different datasets **[1].**

4.1 K-means Clustering Algorithm

In this research, the k-means clustering algorithm has been described as one of the unsupervised learning - data mining algorithm. The objective of this algorithm is to find collections (centroids) in the data, with the number of sets denoted by the variable "k".

The algorithm works iteratively to allocate each data point to one of "k" groups based on the structures that are provided. The data points are gathered based on structure comparison.

The results of the k-means clustering algorithm are:

- The centroids of the "k" clusters, which can be used to label new data
- Labels for the training data (each data point is assigned to a single cluster)

The k-means clustering algorithm uses iterative refinement to produce a final result.

The algorithm inputs are the number of clusters "k" and the dataset. The dataset is a collection of features for each data point. The algorithm begins with initial approximations for the "k" centroids, which can either be randomly generated or randomly selected from the data set.

The algorithm then iterates between two phases:

4.2 Cluster Classification of Data Points:

Each centroid defining one of the clusters. In this step, each data point is assigned to its nearest centroid, based on the squared Euclidean distance.

Theoretically, if k_i is the collection of centroids in set K, then each data point x is assigned to a cluster based on

$$\underset{k_i \in K}{argmin} \, dist(k_i, x)^2$$

where,

$dist(\cdot)$ - the standard Euclidean distance.

Approving the set of data point assignments for each i^th cluster centroid be S_i.

4.3 Recompute Centroids:

Here the centroids are recomputed.

This is done by taking the mean of all data points assigned to that centroid's cluster.

$$k_i = \frac{1}{|s_i|}\sum_{x_i \in s_i} x_i$$

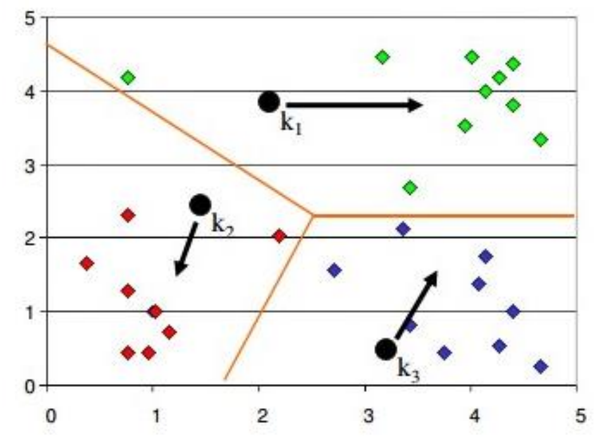


Figure 4: k-means clustering algorithm moves the centres to the means of groups

The algorithm iterates between steps one and two until a stopping criteria is met (i.e., no data points change clusters, the sum of the distances is minimized or some maximum number of iterations is reached); with definite process to converge to a result. There is possibility that the result becoming a local optimum, meaning that assessing more than one run of the algorithm with randomized starting centroids may give a better outcome. This is sometimes viewed as computationally expensive with large datasets.

4.4 Algorithm Work

The algorithm will focus on ED, where four (4) main variables are used.

They are:

1) **N**: Number of points
2) **D**: Dimension of points
3) **K**: Number of groups ("k" clusters or centroids)
4) **I**: Iterations (sequence of outcomes)

We have defined the amount of effort (**WORK**) done by the algorithm as:

$$\mathbf{N} \times \mathbf{D} \times \mathbf{K} \times \mathbf{I}$$

5. APPLICATION DESCRIPTION AND METHODS

To conduct the empirical study of our GPU-accelerated implementation of the k-means clustering algorithm for Advanced Analytics, the understanding of the application design approaches of the implementations of the k-means clustering algorithm must be clearly defined.

In this research, there are **three (3)** application design approaches that was used to develop a GPU-accelerated implementation of the k-means clustering algorithm. They are as followed:

1) **CPU** – Baseline Version
   i. Random Dataset
   ii. Input Dataset
2) **CPU** – Optimized Version
3) **GPU** – Accelerated Verson

5.1 CPU – Baseline Versions

We begin with two (2) types of baseline versions, **Random** and **Input Dataset**.

5.1.1 Random Dataset

Firstly, the description and structure of our initial version defined as **"kmeans_baseline.c".**
The source code **"kmeans_baseline.c"** includes 5 functions including main() function.
The main variables and constant used in code are as followed:

a) **N**: number of points (default value: 1000)
b) **D**: dimension of point (default value: 3)
c) **C**: number of groups (clusters) (default value: 4)
d) **Punts**: dataset (2d integer matrix, i.e., N*D)
e) **Centroides**: centre points of each group (2d matrix, K*D)
f) **PC**: index array of points which assigned to each centroid (N*K)
g) **Sep**: number of points for each group (1d vector with length K)

In the first function, it generates N points which dimension is D at the beginning of algorithm.

-
```
void generatePunts(int *Punts, int N, int D, int K, double *Centroides,
int DIMENSION)
```

Here, DIMENSION means the range which value of point can be represented.
The function rand() returns a **random integral number** in the range between 0 and RAND_MAX.
RAND_MAX is a constant defined in <cstdlib>.

Thus, Punts matrix is randomly filled with an integer in the range between 0 and 999.

Then, initial K points is selected as centre of each groups.

Following is to display initial centre points with a function to assign data points into the closest group:

```
printf("\nCentroides:\n\t");
for(i=0; i<K; i++){
	for(j=0; j<D; j++)
		printf("%3.1f\t",Centroides[i*D+j]);
	printf("\n\t");}
```

- `void PointsToCentroides(int *Punts, int N, int D, int K, double *Centroides, int *PC, int *Sep)`

After that we will, initialize **Sep** vector, i.e., number of points in each group is set to 0.

```
for(i=0; i<K; i++)
	Sep[i]=0;		// Number of assigned points in each class
```

Then, the loop following process for each point.

For i-th point of dataset Punts, calculate all distance between that point and centre point of each group and find group which gives minimum distance (nearest group).

And reassign i-th point to that nearest group and increase number of points of that group.

After this step, each data point is reassigned to its nearest centroid, based on the squared Euclidean distance. The function mmin(·, ·) is used to find minimum distance.

That is, it returns the index which gives minimum distance among K distances that represent Euclidean distance between i-th point and all centroid.

- `int recalcularCentre(int *Punts, int N, int D, int K, double *Centroides, int *PC, int *Sep)`

In this step, the centroids are recomputed. This is done by taking the mean of all data points assigned to that centroid's group. However, if none of centres changed, algorithm is terminated.

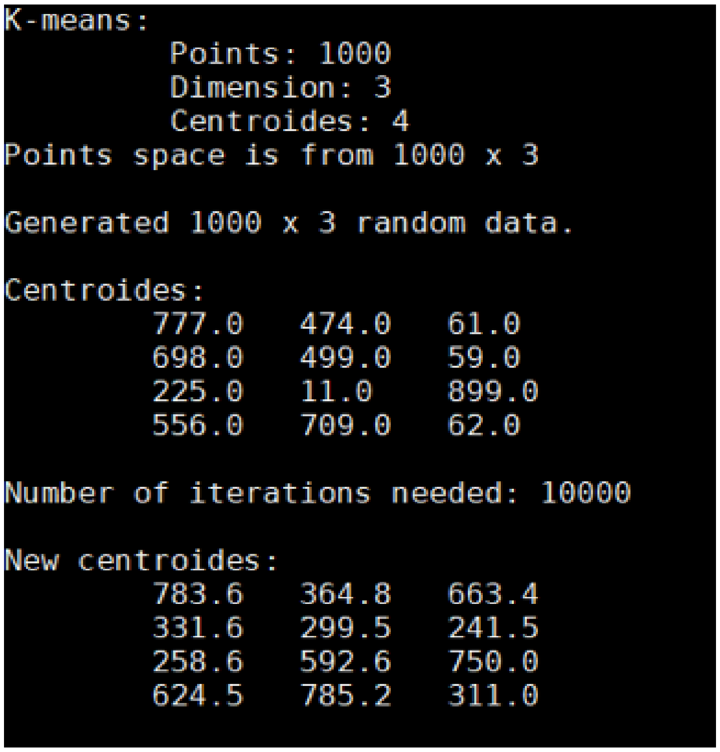
After initializing all variables and constants, the k-mean clustering algorithm is implemented as following:

```
do{
	// reassignment each point to nearest centroids
	PointsToCentroides(Punts, N, D, K, Centroides, PC, Sep);

	// counting iteration round
	cont += 1;

	// check if no changed
	final = recalcularCentre(Punts, N, D, K, Centroides, PC, Sep);
}while(final && cont<200);
```

Iteration terminates when clusters have converged or an iterative count is over limitation.

After the iteration is finished, the program prints out updated centre points.

Screenshot of running this program is as followed:



```
K-means:
        Points: 1000
        Dimension: 3
        Centroides: 4
Points space is from 1000 x 3

Generated 1000 x 3 random data.

Centroides:
        777.0   474.0   61.0
        698.0   499.0   59.0
        225.0   11.0    899.0
        556.0   709.0   62.0

Number of iterations needed: 10000

New centroides:
        783.6   364.8   663.4
        331.6   299.5   241.5
        258.6   592.6   750.0
        624.5   785.2   311.0
```

Figure 5: Sample execution of the program “**kmeans_baseline.c**”

5.1.2 Input Dataset

This program is a restructured version of **“kmeans_baseline.c”** to develop **“kmeans_baseline_input.c”**. The difference of them are reading the selected input dataset from **csv** (comma-separate value) file instead of randomly generating a dataset, as required.

Therefore, here will be a reduced description of the same functions and a representation of structural updates to the previous code, where necessary.

As before, all variables and constants used in code are the same.

N: number of points (default value: 1000)

D: dimension of point (default value: 3)

K: number of groups (clusters) (default value: 4)

Punts: Dataset (2d integer matrix, i.e., N*D)

Centroides: centre points of each group (2d matrix, K*D)

PC: Index array of points which assigned to each centroid (N*K)

Sep: number of points for each group (1d vector with length K)

The source code "**kmeans_baseline_input.c**" includes the following updates:

- Firstly, open data file using input csv file name and numbers of points, dimension and groups.
- If input data doesn't exist then print error message and exit.
- Second, loop with the following iteration:
    - Read one line (for ex, 25.95, 40.13, 28.76, 5.16, 36.38, 26.42, 20.83, 17.72 ...).
    - Get each float value which separated with comma (,).
    - If the number of float values in each line doesn't equal to '**D**' then an error.
    - Put those float values into Punts array.
    - If number of lines > N, program cuts remain lines.
    - Otherwise, N = read lines
    - Then, calculate centres as same as before.
- `int read_CSV_data(char* filename, float* Punts, int N, int D, int K, float* Centroides)`

In function **read_CSVdata()**, the program reads data points from the given .csv files and assigns centres of each group randomly.

```
FILE* fp = fopen(filename, "r");
```

This line means that the open file with named "**filename**" as read-only mode.

If the program successfully to opens the file, it will return the pointer of the file, otherwise, returns NULL.

Here, we used this to check the opening of file:

```
if (!fp) {
        printf("Can't fine input file - %s\n", filename);
        return 1;
           }
```

That is, if the program fails to open file, the program prints an error-message and exits.

If successful, the program reads the data line by line from .csv file; until completed.

Reading one line is implemented by

```
// read first line (NameID, V, A, B, C, GM, KA, CP, IS...)
fgets(line, sizeof(line), fp);
```

where **fp** is a pointer of opened csv file.

Then, we should see that line, for example "NameID, V, A, B, C, GM, KA, CP, IS, ...".

This first line is not needed to enter the data points. So, we iterate the loop from second line.

```
char* token = strtok(line, ",");
```

This is to get sub-string which is separated with comma from line, so we get token = “NameID”.

This token is called “the row name”.

Therefore, we store this string value into a row_name array.

All other strings at line converts to a float number and is put into a data array as follows.

```
// convert string to float value and is sent to Punts array
       Punts[n_total_idx++] = atof(token);
       n_each_in_line += 1; // increase number of read float value
       j += 1;
```

Here, n_each_in_line represents number of values at each line, that is, means dimension of a point.

If this n_each_in_line is not equal to value of <D>, program prints an error-message and exits.

All above process is repeated until the programs reads all line from .csv file and closes the opened file.

And then, as before, it initializes the centre points of all group randomly, print its value and assign data points into the closest group.

Now, **Sep** vector initializes, i.e., number of points in each group is set as 0.

```
       for(i=0; i<C; i++)
              Sep[i]=0; // Number of assigned points in each class
```

Then, loop following process for each point.

For i-th point of dataset Punts, calculate all distance between that point and centre point of each group and find group which gives minimum distance (nearest group).

And reassign i-th point to that nearest group and increase number of points of that group.

After this step, each data point is reassigned to its nearest centroid, based on the squared Euclidean distance. The function mmin(·, ·) is used to find minimum distance.

That is, it returns the index which gives minimum distance among K distances that represent Euclidean distance between i-th point and all centroid.

- `int recalcularCentre(int *Punts, int N, int D, int K, double *Centroides, int *PC, int *Sep)`

In this step, the centroids are recomputed. This is done by taking the mean of all data points assigned to that centroid’s group. However, if none of centres changed, algorithm is terminated.

Firstly, variables of “N”, “D” and “K” are defined.

```
       int N;              //Number of points
       int D;              //Dimensions of the point
       int K;              //Number of centroides
```

Then, this is string variable to represent file name.

```
// string array to be read file name
char* file = (char*)malloc(256);
```

At this point, we check all arguments and set to all variables.

Default number of arguments is 5.

<kmeans_baseline_input> <InputData_NameID.csv> <K> <N> <D>

If all arguments aren't given, the program uses default parameters for testing "InputData_NameID.csv" as follows.

```
// if all parameters aren't given, set as default parameter
if (argc < 5) {
      // Default parameters
      strcpy(file, "InputData_NameID.csv");
      K = 4;
      N = 1000;
      D = 3;
printf("Usage: %s<InputData_NameID.csv>, <K=Number of Clusters>, <N=Number of
Rows>, <D=Number of Columns>\n", argv[0]);
printf("We now use default parameters with InputData_NameID.csv, K=%d, N=%d,
D=%d\n", K,N,D);

            }
```

Then, allocate memory for all variables used in the program.

Before starting the k-mean algorithm, read data from csv file and initialize centre points of all groups.

```
// read input data from csv file
if(read_CSV_data(file, Punts, N, D, K, Centroides) > 0)
      exit(1);
```

After the program has initialized and read all variables and constants, the k-mean clustering algorithm is implemented as following:

```
do{
      // reassignment each point to nearest centroids
      PointsToCentroides(Punts, N, D, K, Centroides, PC, Sep);

      // counting iteration round
      cont += 1;

      // check if no changed
      final = recalcularCentre(Punts, N, D, K, Centroides, PC, Sep);
}while(final && cont<200);
```

When the implementation of the algorithm is finished, the program prints out the new centre points.

The program then stores the result into "result.csv" as a storage file.

Firstly, we will get the first line of input .csv file as "NameID" and store it in line.

Then, we created result file with named “result.csv” as write-only mode.

From now, we can write line by line as string value to this file.

An example of the contents of values to be stored into file are as followed:

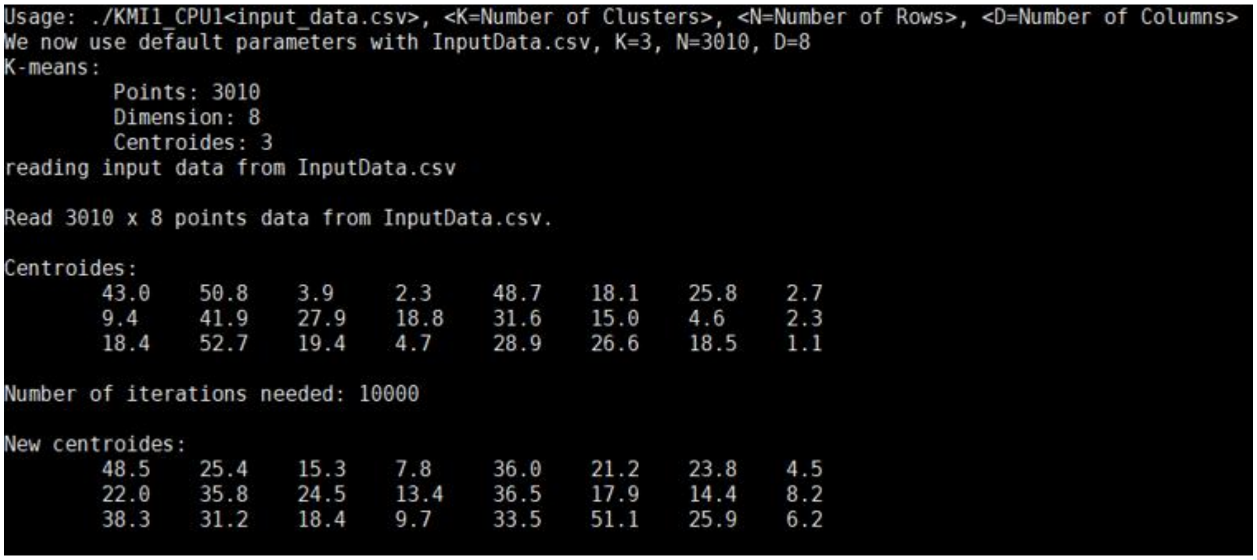
- Number of groups.
  Number of clusters: 3

- Number of points at each group – for instance:
  Cluster-1 : 1098 elements
  Cluster-2 : 1401 elements
  Cluster-3 : 511 elements

- NameID and points which was clusterd into each group for all groups.
  Cluster – 1
  NameID,V,A,B,C,GM,KA,CP,IS,
  35,37.7,34.5,23.5,4.4,35.3,20.3,19.8,2.2,
  36,51.0,20.7,12.5,8.0,37.5,25.8,44.8,0.6,
  37,47.0,26.8,21.8,2.5,36.3,26.9,22.0,2.5,
  45,53.2,26.4,14.4,5.0,44.3,21.7,13.6,2.8,
  ... ...
  Cluster – 2
  NameID,V,A,B,C,GM,KA,CP,IS,
  ....

Finally, close result file and free all allocated memory.

Below is a screenshot of running the input program (with a sample real data set) as followed:



```
Usage: ./KMI1_CPU1<input_data.csv>, <K=Number of Clusters>, <N=Number of Rows>, <D=Number of Columns>
We now use default parameters with InputData.csv, K=3, N=3010, D=8
K-means:
        Points: 3010
        Dimension: 8
        Centroides: 3
reading input data from InputData.csv

Read 3010 x 8 points data from InputData.csv.

Centroides:
        43.0    50.8    3.9     2.3     48.7    18.1    25.8    2.7
        9.4     41.9    27.9    18.8    31.6    15.0    4.6     2.3
        18.4    52.7    19.4    4.7     28.9    26.6    18.5    1.1

Number of iterations needed: 10000

New centroides:
        48.5    25.4    15.3    7.8     36.0    21.2    23.8    4.5
        22.0    35.8    24.5    13.4    36.5    17.9    14.4    8.2
        38.3    31.2    18.4    9.7     33.5    51.1    25.9    6.2
```

Figure 6: Sample execution of the program “**kmeans_baseline_input.c**”

5.2 CPU – Optimized Version

At this point, the program has fulfilled execution requirements needed for the problem at hand. This program is a restructured version of **"kmeans_baseline_input.c"** that was developed as an optimized version called **"kmeansFINAL.c"**. The difference between them is that the program transformation technique called "code optimization" was be used to improve the intermediate "**baseline input version"** by developing it to consume fewer resources (i.e. CPU, Memory); so that it improves the speed and performance of the program. Therefore, here the program optimization process must focus on performance optimization techniques. The two (2) optimization techniques discovered are as followed:

A) **Machine Independent Optimization** (used to improve the code configurations or structure for an enhanced execution, which does not consist of any CPU registers, memory locations or devices)

B) **Machine Dependent Optimization** (used to improve the code that consist of any CPU registers, memory devices for the selected machine architecture and to take full advantage of memory hierarchy) **[26]**.

In this research, we use **Machine Independent Optimization** that focuses more on the improving "**baseline input version**" configurations or structure for an enhanced execution, as required. After studying the program and reorganizing its entire structure (i.e. compiler optimization, data types, pipelines, memory leaks etc.) the following improved techniques were used:

i. **Loop fusion** (to combine multiple loops into a single loop to improve run-time speed and reduce loop overhead)

ii. **Loop-invariant code motion** (used to move statements or expressions outside the body of a loop without affecting the program semantics),

iii. **Loop nest optimization** (used to improve locality of reference or locality parallelization to reduce memory access latency etc.),

iv. **Loop Unrolling** (used to increase speed by reducing or eliminating instructions that control program loops)

v. **Strength Reduction** (used to remove expensive operations or executions with simpler ones) **[26]**.

The optimized program improvements were designed in three (3) parts. They are as followed:

1) **Input** (to generates N - points with D - dimensions randomly and makes the choice of each centroid),
2) **Computation** (to update the center for all the centroids, returns the position (centroid number) where the minimum value of a vector is found and assign each point to the nearest centroid)
3) **Output/Main** (to preform the output results via the kmeans clustering algorithm).

5.3 GPU – Accelerated Version

At this point in the research, the program has fulfilled all the optimized and relevant parallelism requirements needed for data transfer to the GPU. This program is a restructured version of **"kmeansFINAL.c"** that was altered as a GPU-accelerated implementation called **"kmeansACC.c"**. The difference of them is the insertion of the OpenACC Directives in selected parts of the program. OpenACC works with three levels of parallelism via gang, worker and vector. When executing a selected computational region on the GPU, one or more gangs are launched, each with one or more workers, where each worker uses the SIMD (Single instruction, multiple data) vector execution ability with one or more vector line of data flow. Therefore, we developed the relevant parts of the program that completes the data pipeline, performance optimization techniques and simplify parallel programming process (i.e. OpenACC Directives) of heterogeneous CPU/GPU system.

We will now explain the GPU-accelerated implementation, the OpenACC Directives that were inserted in the most computationally expensive parts of **"kmeansFINAL.c"** for high-level data management and optimizing loops in **"kmeansACC.c"**.

Firstly, the "**kmeansFinal.c**" code was modified to include OpenACC directives. Analytically, the **"#pragma acc data copyin(Punts[N*D]) copyout(Labels[N]) copy(Centroides[K*D])**" and **"#pragma acc data create(New_Centr[K*D]) copyout(Sep[K])**" were inserted inside the "main" function before the "do loop", in order to allocate memory (where necessary), move the contents of reassignment each points to nearest centroids from host to GPU and copies data to the host when exiting to the function region.

Furthermore, the "**#pragma acc data present(Punts,Labels,Centroides,New_Centr,Sep) copy(pR[1])"** was inserted inside the "**PointsToCentroides**" function before the first loop to inform the compiler that data is already present on the device with allocated memory on GPU, copy data from host to GPU (device) and copy data to the host when exiting to the function region. The "**#pragma acc parallel loop gang vector_length(128)"** was put in front of the outmost loop and two individual internal outmost loops. These parallel regions will utilize thread blocks, each with 128 threads, where each thread executes one iteration of the loops.

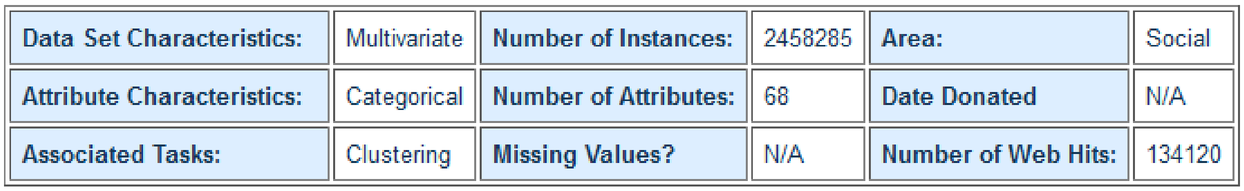
Also, the **"#pragma acc loop vector**" was place before the innermost for loops of each individual **"#pragma acc parallel loop gang vector_length(128)**". The use of **acc parallel** provides massive parallelization and identifies each loop to run in parallel. This directive basically tells the compiler to analyze all the code that is assigned to. As a result, the compiler "understands" the code in the annotated accelerated region and does all the parallelizing of selected parts of the code provided by us.

These changes prepares the GPU-accelerated program "**kmeansACC.c**" for our official input data.

6. DATA MODIFICATION AND OVERVIEW

6.1 Data Description

As discussed earlier, input data will be an official real dataset that was extracted using a Data Extraction System and will fit on the GPU memory.
The USCensus1990raw data set was collected as part of the 1990 census from the (U.S. Department of Commerce) Census Bureau website at http://www.census.gov.



Data Set Characteristics:	Multivariate	Number of Instances:	2458285	Area:	Social
Attribute Characteristics:	Categorical	Number of Attributes:	68	Date Donated	N/A
Associated Tasks:	Clustering	Missing Values?	N/A	Number of Web Hits:	134120

Figure 7: US Census Data (1990) Data Set Description, Retrieved Web. 19 Apr. 2019. **[29, 30]**

The data set was extracted using the Data Extraction System found at http://dataferrett.census.gov.
The data set is a 1% sample of the Public Use Microdata Samples (PUMS) person records drawn from the full 1990 census sample. Figure 7 shows that the data set contains 68 categorical attributes with a sequence of operations via Randomization and Selection of Attributes. It also contains a total of 2,458,285 number of instances or data points.

6.2 Data Layout

Below shows the original dataset layout:

caseid,dAge,dAncstry1,dAncstry2,iAvail,iCitizen,iClass,dDepart,iDisabl1,iDisabl2,……

10000,5,0,1,0,0,5,3,2,2,1,0,1,0,4,3,0,2,0,0,1,0,0,0,0,10,0,1,0,1,0,1,4,2,2,3,0,2……

10001,6,1,1,0,0,7,5,2,2,0,0,3,0,1,1,0,1,0,0,0,0,1,0,0,4,0,2,0,0,0,1,4,1,2,2,0,2……..

10002,3,1,2,0,0,7,4,2,2,0,0,1,0,4,4,0,1,0,1,0,0,0,0,0,1,0,2,0,4,0,10,4,1,2,4,0,2……

10003,4,1,2,0,0,1,3,2,2,0,0,3,0,3,3,0,1,0,0,0,0,0,0,1,4,0,2,0,2,0,1,4,1,2,2,0,2,0…...

10004,7,1,1,0,0,0,0,2,2,0,0,3,0,0,0,0,0,0,0,0,1,0,0,0,0,0,2,2,0,0,0,4,1,2,0,0,2,0…...

……..

6.3 Data Input and Readability

To conduct our experimental study, the dataset was reduced to 40.68%, to exactly 1 million data points for advanced analytics insight and industry standard readability via .CSV; which can be regenerated to open in Microsoft Excel from the Microsoft Office suite of software.

6.4 Data Density Heatmap

After selecting the required dataset, we conducted a “scaling of data” due to the source dataset spanning in different ranges within an unstructured manner. A graphical representation of dataset representing the density of dots in a map was demonstrated.

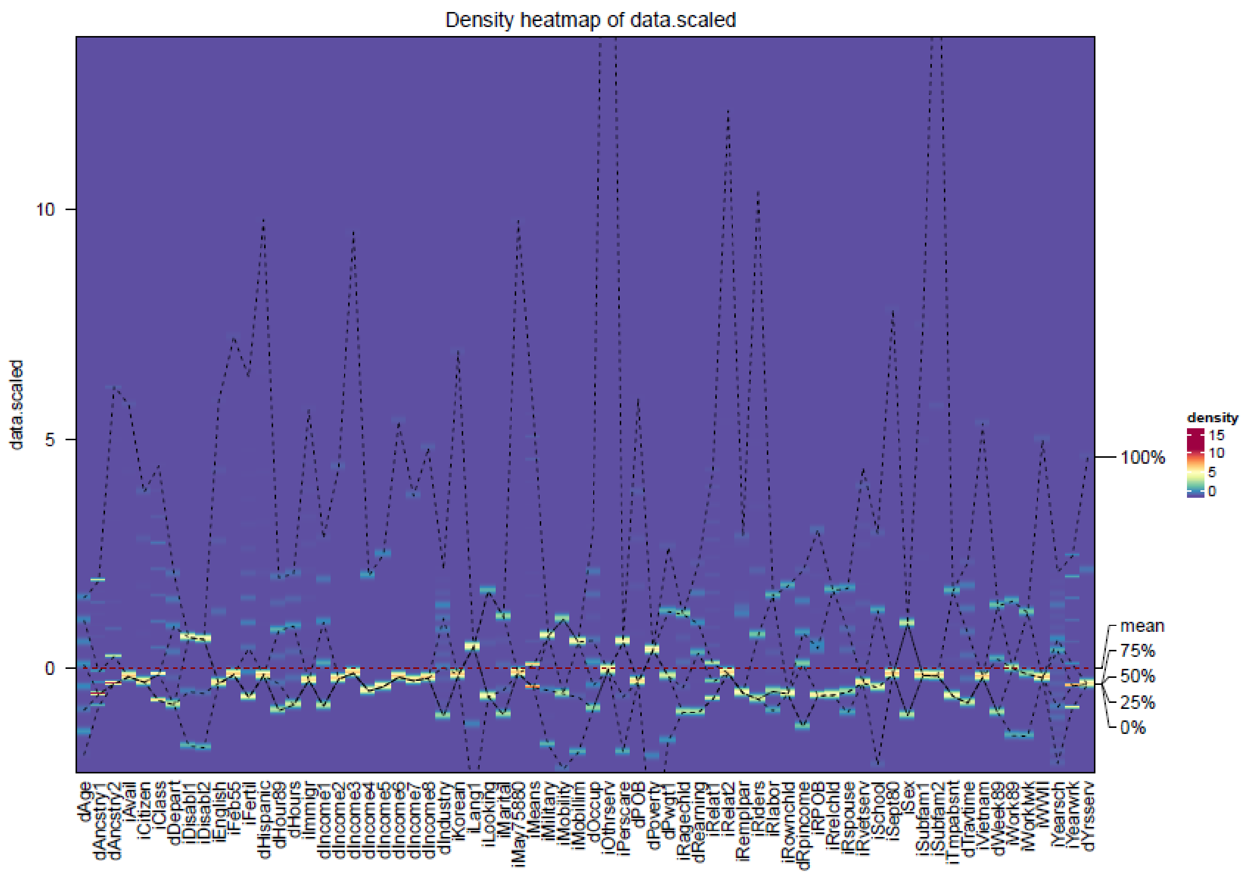


Figure 8: Density Heatmap of the US Census Data (1990) Data Set Normalized

Figure 8 shows the perceive density of points using a scale function to normalize the data points.

7. COMPUTER ARCHITECTURE AND RESOURCE MANAGEMENT

All executions were conducted in a Computer Architecture Lab (via heterogeneous cluster for executing batch jobs) and Queue System (via execution queues (or partitions) configured using SLURM - Simple Linux Utility for Resource Management) as a jobs manager.

Let's now define the accessible queues and nodes used:

test.q: contains only the aolin-login node

- This node provides up to 4 threads (4 cores, each core with 1 H/W threads) and no GPU. This partition was used for testing of all submission scripts.

cuda.q: contains nodes aolin[11-15], aolin17, aolin19, aolin21, aolin23, aolin24, and aomaster

- These nodes contain GPUs and some nodes contain the higher number of CPU cores.

The characteristics of the computation queues nodes in the cluster used:

aolin24:

- Processor: Intel Core i7-3770 @ 3.40GHz (4 cores, 2 threads x core: total of 8 H/W threads)
- L3 cache: 8MB; Memory: 8 GB at 1067 MHz
- NVIDIA GeForceGTX 1080 Ti (3584 cores)

The submission of the parallel GPU jobs will be executed on the ***aolin24*** node, with the "***NVIDIA GeForceGTX 1080 Ti***" GPU device, which is autonomous of the aolin-login node via SLURM.

Also, for all parallel GPU jobs, the submission must be sent to the ***cuda.q*** queue with a special option to specify the number of GPUs reserved for the execution job via ***Gres=gpu:name_GPU:number*** or ***Gres=gpu:number*** (i.e. do not select a GPU device).

In this case, the following commands had to be typed before each session:

```
#!/bin/bash -l
#
#SBATCH --job-name=ML_KMs
#SBATCH -N 1 # number of nodes
#SBATCH -n 2 # number of cores TOTALES
#SBATCH --partition=cuda.q
#SBATCH --nodelist=aolin24
#SBATCH --gres=gpu:1
```

GNU Compiler toolset was installed:	***module add gcc/8.2.0***
The PGI toolset was installed:	***module load pgi/18.4***
The CUDA toolset was installed:	***module load pgi/18.4***
Show info about available GPUs:	***nvidia-smi***

8. APPLICATION PERFORMANCE ANALYSIS AND RESULTS

In this research, we want to measure the application performance and study the results of the Multi-Core (MCore) CPUs and GPU accelerated implementations (via OpenACC directives). We will conduct two (2) distinctive experiments,

1) "K" Clusters, ("K" number of groups with all other variables remain constant),
2) "D" Multidimensions ("D" dimensions of points with all other variables remain constant),

The focus will be on application execution, utilization of the GPU device, scalability and speedup. Since, we are assuming a liner movement (according to Section 4.1.3), we would like to understand the importance of small vs large problems of our accelerated application for big data problems.

8.1 Compiling, Building and Running the Application

All codes, for Multi-Core (MCore) CPUs were compiled with the command: ***pgcc -fast -acc -ta=multicore -Minfo=all kmeansACC.c -o codename.*** For the GPU-accelerated implementations (via OpenACC directives) were compiled with the command: ***pgcc -fast -acc -ta=tesla -Minfo=all kmeansACC.c -o codename.*** Each execution had ***200*** iterations with the use of ***perf stat*** and ***pgprof/ nvprof*** in order to measure basic performance metrics. Every code is resulted from the previous one, with some alterations, and it has been checked that provides correct results.

8.2 Application First Execution – Small Problem

For advanced analytics insight and results industry standard readability, we now display how the GPU-accelerated application executes, using the following input variables:

N: number of points	= **1,000,000** (1 million)
D: dimension of point	= **68**
K: number of groups (k clusters)	= **2**

As discussed before, we present the usual output .CSV file of the k-means clustering problem from our sample of the USCensus1990raw data set that will fit on the GPU memory:

```
Number of clusters: 2
        Cluster-1: 750007 elements
        Cluster-2: 249993 elements
```

```
Cluster - 1
caseid,dAge,dAncstry1,dAncstry2,iAvail,iCitizen,iClass,dDepart,iDisabl1,iDisabl2,……
10000,5.0,0.0,1.0,0.0,0.0,5.0,3.0,2.0,2.0,1.0,0.0,1.0,0.0,4.0,3.0,0.0,2.0,0.0,0.0……
10001,6.0,1.0,1.0,0.0,0.0,7.0,5.0,2.0,2.0,0.0,0.0,3.0,0.0,1.0,1.0,0.0,1.0,0.0,0.0……
10002,3.0,1.0,2.0,0.0,0.0,7.0,4.0,2.0,2.0,0.0,0.0,1.0,0.0,4.0,4.0,0.0,1.0,0.0,1.0……
```

```
10003,4.0,1.0,2.0,0.0,0.0,1.0,3.0,2.0,2.0,0.0,0.0,3.0,0.0,3.0,3.0,0.0,1.0,0.0,0.0……
10004,7.0,1.0,1.0,0.0,0.0,0.0,0.0,2.0,2.0,0.0,0.0,3.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0……
10007,4.0,1.0,2.0,0.0,0.0,6.0,0.0,2.0,2.0,0.0,0.0,4.0,0.0,5.0,5.0,0.0,2.0,1.0,0.0……
……
Cluster - 2
caseid,dAge,dAncstry1,dAncstry2,iAvail,iCitizen,iClass,dDepart,iDisabl1,iDisabl2,……
10005,1.0,1.0,2.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0……
10006,1.0,1.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0……
10012,1.0,1.0,2.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0……
10020,1.0,2.0,2.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0……
10024,1.0,1.0,2.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0……
10026,1.0,3.0,2.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0……
……
```

The most relevant performance metrics (via **perf stat**) for the "**kmeansACC.c**" code are the elapsed execution time (10.09 seconds time elapsed), the total number of executed machine instructions (57.104 Giga), the IPC rate (1.55 insns per cycle) and total utilization rate (0.969 CPUs utilized).

8.3 GPU Device Activity

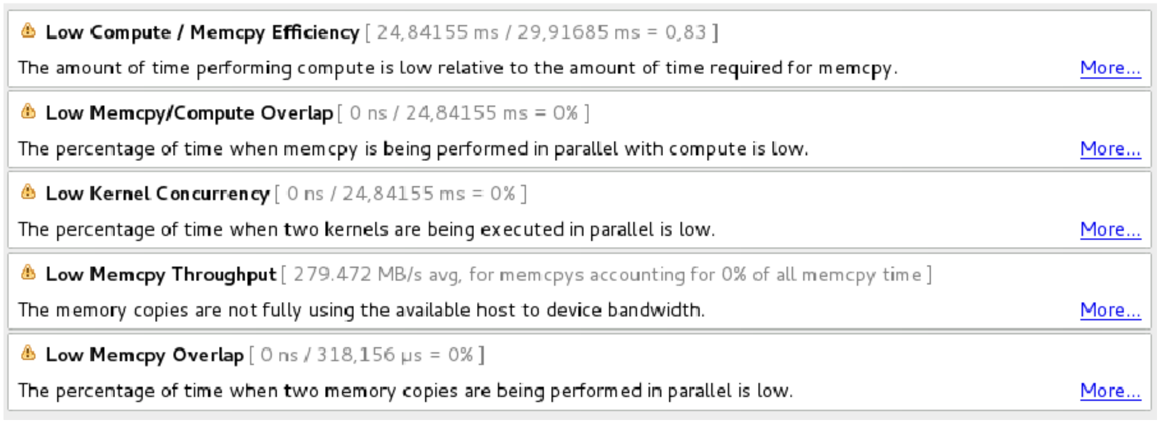
Continuing with the k-means clustering problem from our sample of the USCensus1990raw data set, where N = 1M, D = 68, K = 2 and I = 200, the command "**pgprof**" was used to measure the execution time of each kernel, displaying the GPU activity as following:

Time (%)	Time	Calls	Avg	Min	Max	Name
99.12%	**3.62860s**	**200**	**18.143ms**	**17.782ms**	**22.388ms**	**PointsToCentroides_121_gpu**
0.85%	31.028ms	18	1.7238ms	896ns	1.9869ms	[CUDA memcpy HtoD]
0.01%	471.25us	203	2.3210us	736ns	315.05us	[CUDA memcpy DtoH]
0.01%	426.10us	200	2.1300us	1.9520us	3.2000us	PointsToCentroides_215_gpu
0.01%	225.57us	200	1.1270us	1.0880us	2.0810us	PointsToCentroides_111_gpu
0.00%	112.33us	200	561ns	513ns	832ns	[CUDA memset]

Table 1: **pgprof** results of the k-means clustering problem where N = 1M, D = 68, K = 2 and I = 200

Table 1 shows the kernel assignment each point to the nearest centroid and updating the center for all the centroids takes 99.12% of the execution time on GPU and the kernel reassignment each points to nearest centroids memory allocation and copying (in/out) of Punts, Centroides and Labels takes 0.01% with the kernel final memory allocation, data creation of New_Centr and copies data out of Sep to the host takes only 0.01% of the execution time. Total execution time of GPU device was 4.03 seconds. In the first row, kernel "**PointsToCentrodies_121_gpu**", utilized the GPU 99.12% of the time from our 200 calls with total time of 3.62860s, average time at 18.143ms, maximum time at 22.388ms and minimum time at 17.1782ms.

Afterwards, the NVIDIA Visual Profiler (**nvvp**) tool was used to provide information about understanding the performance of the GPU application. The results of the GPU usage are shown below:



Low Compute / Memcpy Efficiency [24,84155 ms / 29,91685 ms = 0,83]
The amount of time performing compute is low relative to the amount of time required for memcpy. More...

Low Memcpy/Compute Overlap [0 ns / 24,84155 ms = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

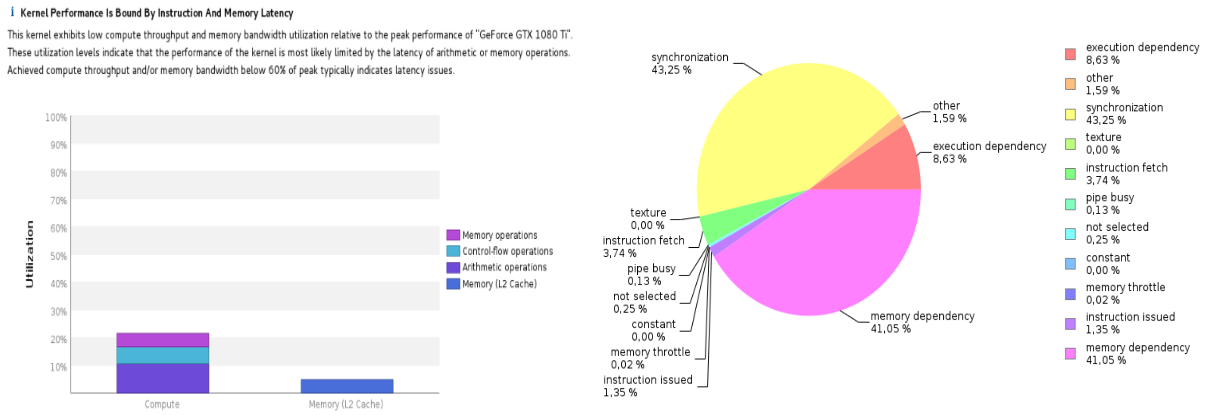
Low Kernel Concurrency [0 ns / 24,84155 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Memcpy Throughput [279.472 MB/s avg, for memcpys accounting for 0% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Low Memcpy Overlap [0 ns / 318,156 μs = 0%]
The percentage of time when two memory copies are being performed in parallel is low. More...

Figure 9: Data movement and Concurrency results shown by the **nvvp** tool.

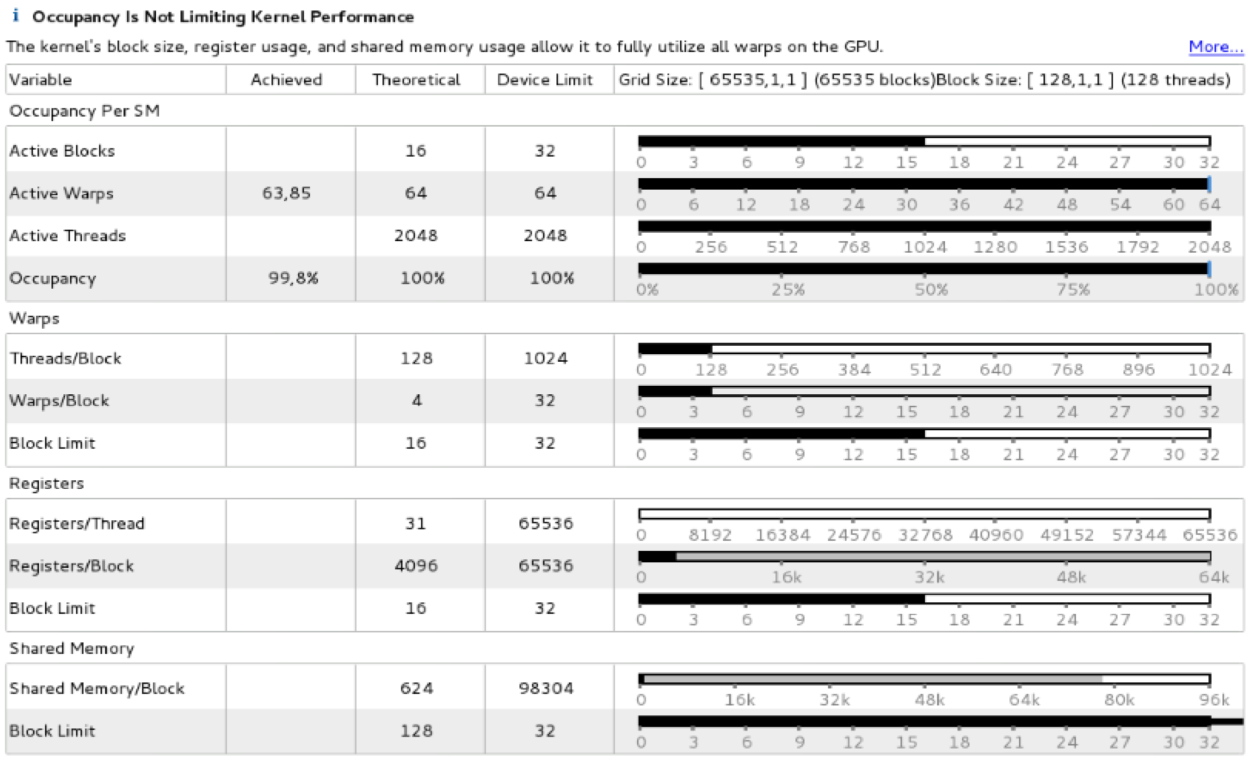
When examining the kernel (**PointsToCentroides_121_gpu**) that assigns each point to the nearest centroid and updating the center for all the centroids, which has a rank of 100 in the optimization importance level, the results obtained are the following:



Kernel Performance Is Bound By Instruction And Memory Latency
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1080 Ti". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Figure 10: Kernel performance and profile pie chart results shown by the **nvvp** tool.

The compute units are 22% utilized, while the memory units are around 5% utilized. The memory resource with the higher utilization percentage is the L2 Cache memory.



i Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [65535,1,1] (65535 blocks)Block Size: [128,1,1] (128 threads)
Occupancy Per SM				
Active Blocks		16	32	0 3 6 9 12 15 18 21 24 27 30 32
Active Warps	63,85	64	64	0 6 12 18 24 30 36 42 48 54 60 64
Active Threads		2048	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	99,8%	100%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		128	1024	0 128 256 384 512 640 768 896 1024
Warps/Block		4	32	0 3 6 9 12 15 18 21 24 27 30 32
Block Limit		16	32	0 3 6 9 12 15 18 21 24 27 30 32
Registers				
Registers/Thread		31	65536	0 8192 16384 24576 32768 40960 49152 57344 65536
Registers/Block		4096	65536	0 16k 32k 48k 64k
Block Limit		16	32	0 3 6 9 12 15 18 21 24 27 30 32
Shared Memory				
Shared Memory/Block		624	98304	0 16k 32k 48k 64k 80k 96k
Block Limit		128	32	0 3 6 9 12 15 18 21 24 27 30 32

Figure 11: Varying block size, register count and shared memory usage shown by the **nvvp** tool.

The utilization of both resources is low (both compute and memory resources). As a result, it can be assumed that the performance is bounded by latency, and specifically, as introduced by the profiler, instruction and memory latency. Instruction and memory latency limitation indicates that the GPU does not have enough work because instruction execution is stalling excessively.

The pie chart in Figure 10 shows that the three primary latency issues in this kernel are synchronization (43.25%), execution dependency (8.63%) and memory dependency (41.05%), which counts for a rank of 92.93 (92.93%) in the optimization importance.

Furthermore, Figure 11 shows evidently that almost full parallelism running on the GPU have been achieved. This have been confirmed by the NVIDIA Visual Profiler that occupancy is not limiting the kernel performance due to the block size, register usage and shared memory, which is fully utilizing all warps on the GPU. The achieved Occupancy is 99.8%, while the Theoretical is 100%. Occupancy is a measure of how much parallelism is running on the GPU versus how much theoretically could be running.

8.4 Application Scalability

The scalability of the algorithm using Multi-Core CPUs and combine computation between CPU and GPU in a heterogeneous system with performance analysis process of multiple 'K' clusters and 'D' multidimensional experimentations will now be presented.

Tableau Software Version 2019.1 (Build Number: 20181.18.0416.1335) will be used for data visualization results of all our experimentations.

8.4.1 'K' Cluster Experimentations

We begin with the performance of selected 'K' clusters (with N = 1M, D = 68, K and I = 200) results.

8.4.1.1 Multi-Core CPUs Performance (Work/Time)

Figure 12 shows the data visualization of performance of selected "K" Clusters.

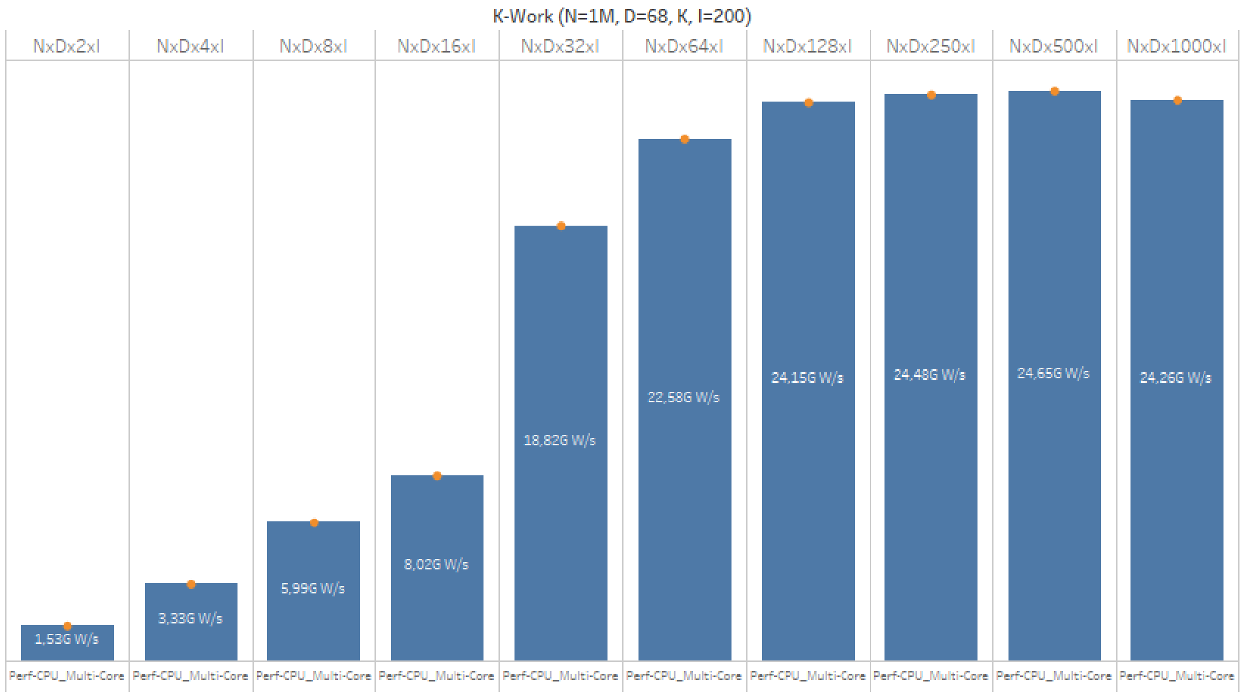


Figure 12: Multi-Core performance of selected 'K' clusters (with N = 1M, D = 68, K and I = 200)

8.4.1.2 GPU (via OpenACC directives) Performance (Work/Time)

Figure 13 shows the data visualization of performance of selected "K" Clusters:

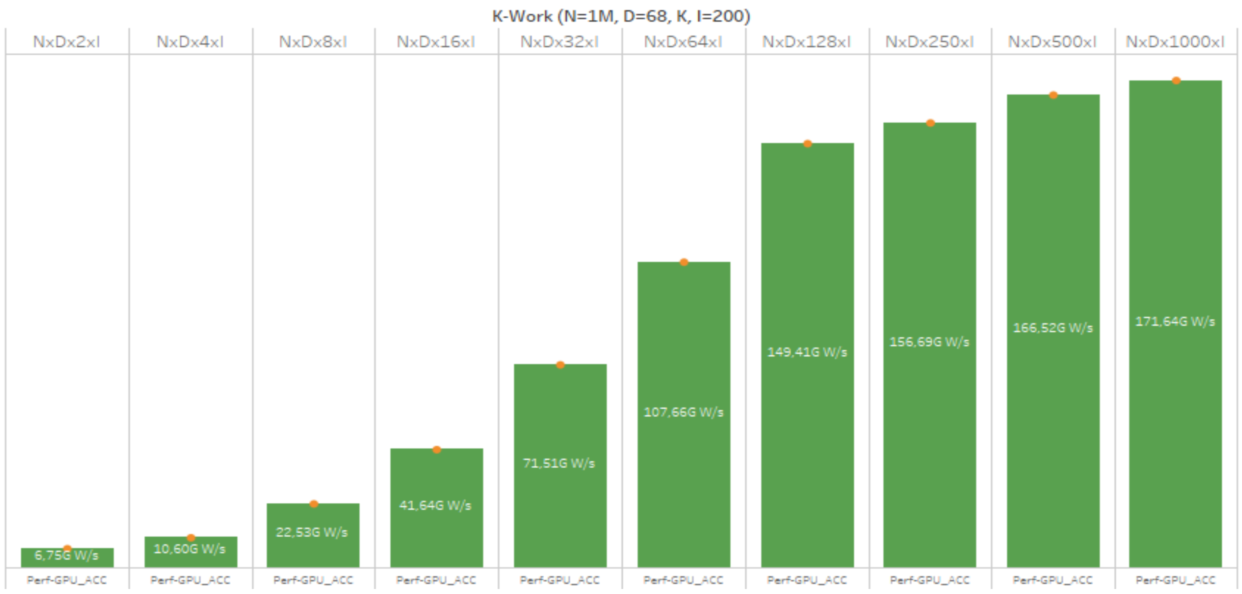


Figure 13: GPU performance of selected 'K' clusters (with N = 1M, D = 68, K and I = 200)

8.4.2 "D" Multidimensional Experimentations

Now, we display the performance of selected 'D' multidimensional experimentations (with N = 1M, D, K = 1000 and I = 200) results.

8.4.2.1 Multi-Core CPUs Performance (Work/Time)

Figure 14 shows the data visualization of performance of selected 'D' multidimensional

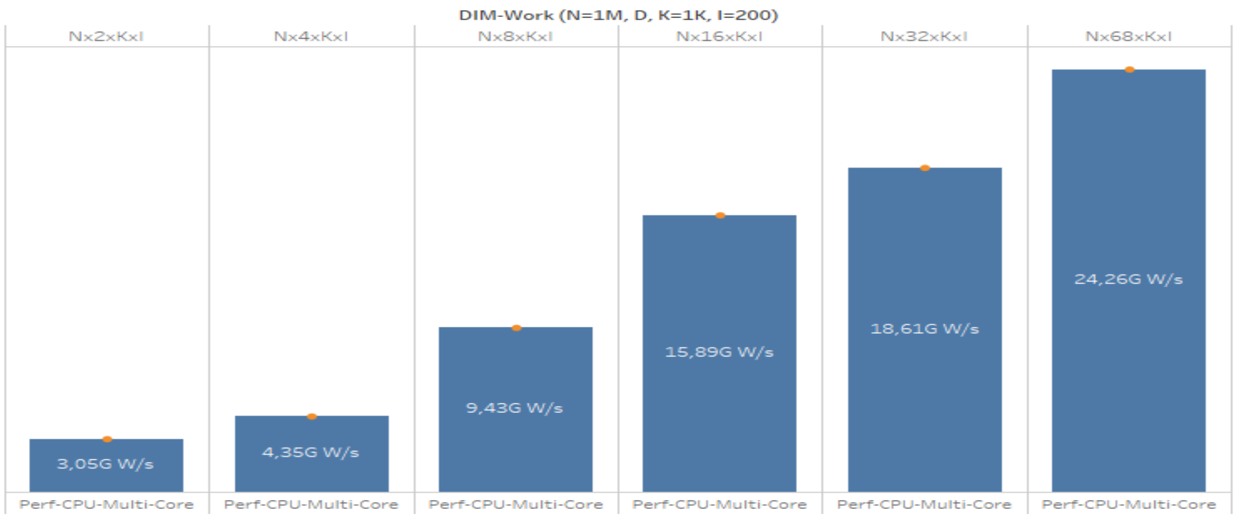


Figure 14: Multi-Core performance of selected 'D' multidimensional (with N = 1M, D, K = 1000 and I = 200)

8.4.2.2 GPU (via OpenACC directives) Performance (Work/Time)

Figure 15 shows the data visualization of performance of selected ‘D’ multidimensional:

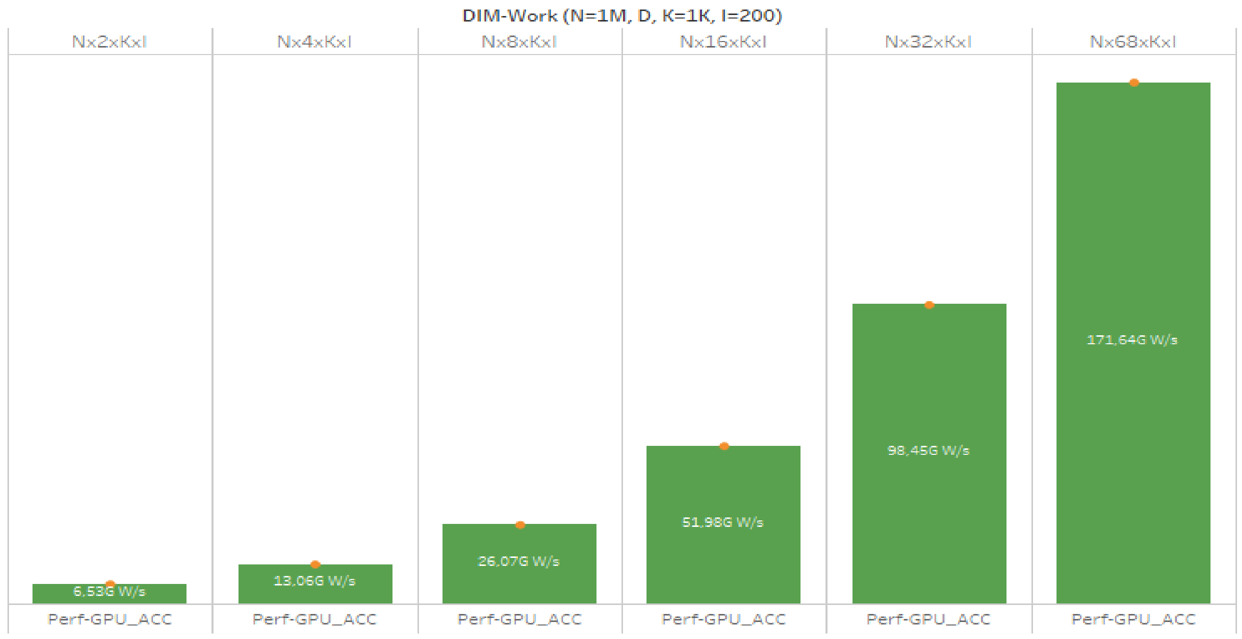


Figure 15: GPU performance of selected ‘D’ multidimensional (with N = 1M, D, K = 1000 and I = 200)

As we can see from Figure 12 to 15, all experiments for “K” Clusters (K = 2, 4, 8, 16, 32, 128, 250, 500 and 1000) and ‘D’ multidimensional (D = 4, 8, 16, 32 and 68) have a visible increase in application performance processing the same problem. Within 80% of the experiments (from K = 64 to 1000), studying figure 12 and 13, the application seems to reach a breaking point in performance levels. Figure 14 and 15 show a steady increase of less than 50% of the application performance for selected ‘D’ multidimensional experiments.

8.5 Application Final Execution – Large Problem

It is noticeable that our application performs well for very large problems modifying the program input variables and increasing the input dataset structure. This tell us that our GDP-accelerated application is designed to solve big data problems to effectively utilize its full capabilities and computational architecture. This can be seen in the GPU activity from our application base execution where N = 1M, D = 68, K = 2 and I = 200.

Let us now study the GPU activity of our final experiment with the k-means clustering problem from our sample of the USCensus1990raw data set, where N = 1M, D = 68, K = 1000 and I = 200.

The most relevant performance metrics (via **perf stat**) for the "**kmeansACC.c**" code are the elapsed execution time (82.27 seconds time elapsed), the total number of executed machine instructions (559.56 Giga), the IPC rate (1.75 insns per cycle) and total utilization rate (0.999 CPUs utilized).

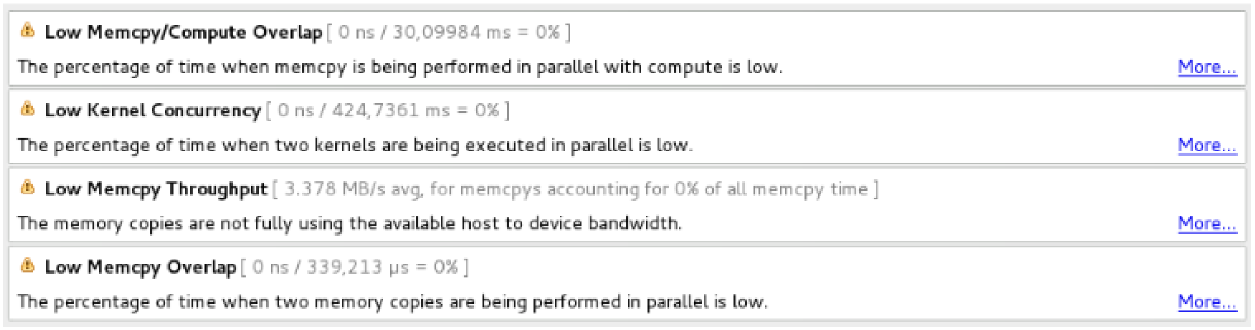
The command "**pgprof**" was used to measure the execution time of each kernel, displaying the GPU activity as following:

Time (%)	Time	Calls	Avg	Min	Max	Name
99.96%	**77.7868s**	**200**	**388.93ms**	**382.43ms**	**636.76ms**	**PointsToCentroides_121_gpu**
0.04%	29.804ms	18	1.6558ms	22.689us	1.9623ms	[CUDA memcpy HtoD]
0.00%	1.2070ms	200	6.0350us	5.7920us	8.2880us	PointsToCentroides_215_gpu
0.00%	498.55us	203	2.4550us	768ns	314.73us	[CUDA memcpy DtoH]
0.00%	463.76us	200	2.3180us	2.2400us	3.8080us	PointsToCentroides_111_gpu
0.00%	114.34us	200	571ns	544ns	864ns	[CUDA memset]

Table 2: **pgprof** results of the k-means clustering problem where N = 1M, D = 68, K = 1000 and I = 200

Table 2 shows the kernel assignment each point to the nearest centroid and updating the center for all the centroids takes 99.96% of the execution time on GPU and the kernel reassignment each points to nearest centroids memory allocation and copying (in/out) of Punts, Centroides and Labels takes less than 0.01% with the kernel final memory allocation, data creation of New_Centr and copies data out of Sep to the host takes less 0.01% of the execution time. Total execution time of GPU device was 79.23 seconds. In the first row, kernel "**PointsToCentrodies_121_gpu**", utilized the GPU 99.96% of the time from our 200 calls with total time of 77.7868s, average time at 388.93ms, maximum time at 382.43ms and minimum time at 636.76ms.

Afterwards, the NVIDIA Visual Profiler (**nvvp**) tool was used to provide information about the performance of the GPU application. The results of the GPU usage are shown below:



Low Memcpy/Compute Overlap [0 ns / 30,09984 ms = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

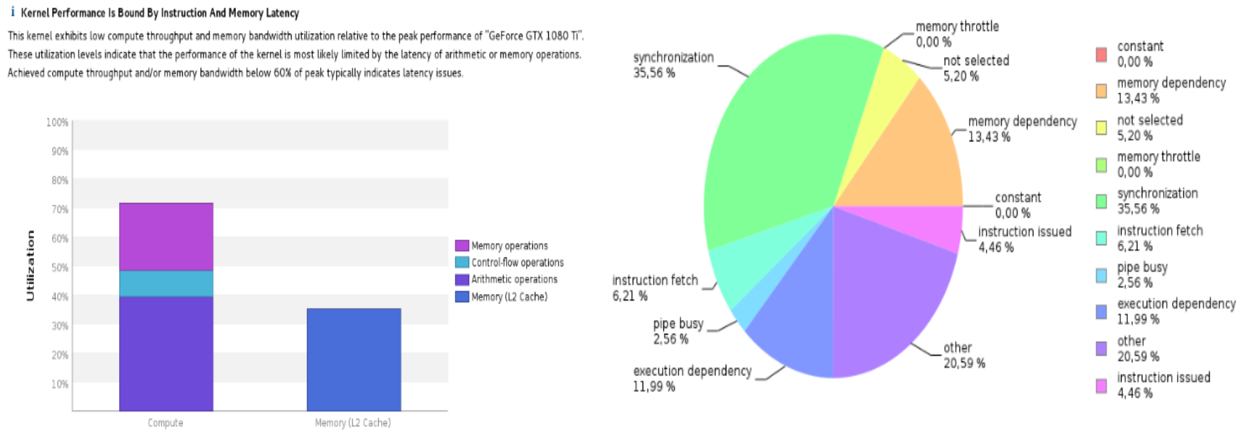
Low Kernel Concurrency [0 ns / 424,7361 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Memcpy Throughput [3.378 MB/s avg, for memcpys accounting for 0% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Low Memcpy Overlap [0 ns / 339,213 μs = 0%]
The percentage of time when two memory copies are being performed in parallel is low. More...

Figure 16: Data movement and Concurrency results shown by the **nvvp** tool.

When examining the kernel (**PointsToCentroides_121_gpu**) that assigns each point to the nearest centroid and updating the center for all the centroids, which has a rank of 100 in the optimization importance level, the results obtained are the following:

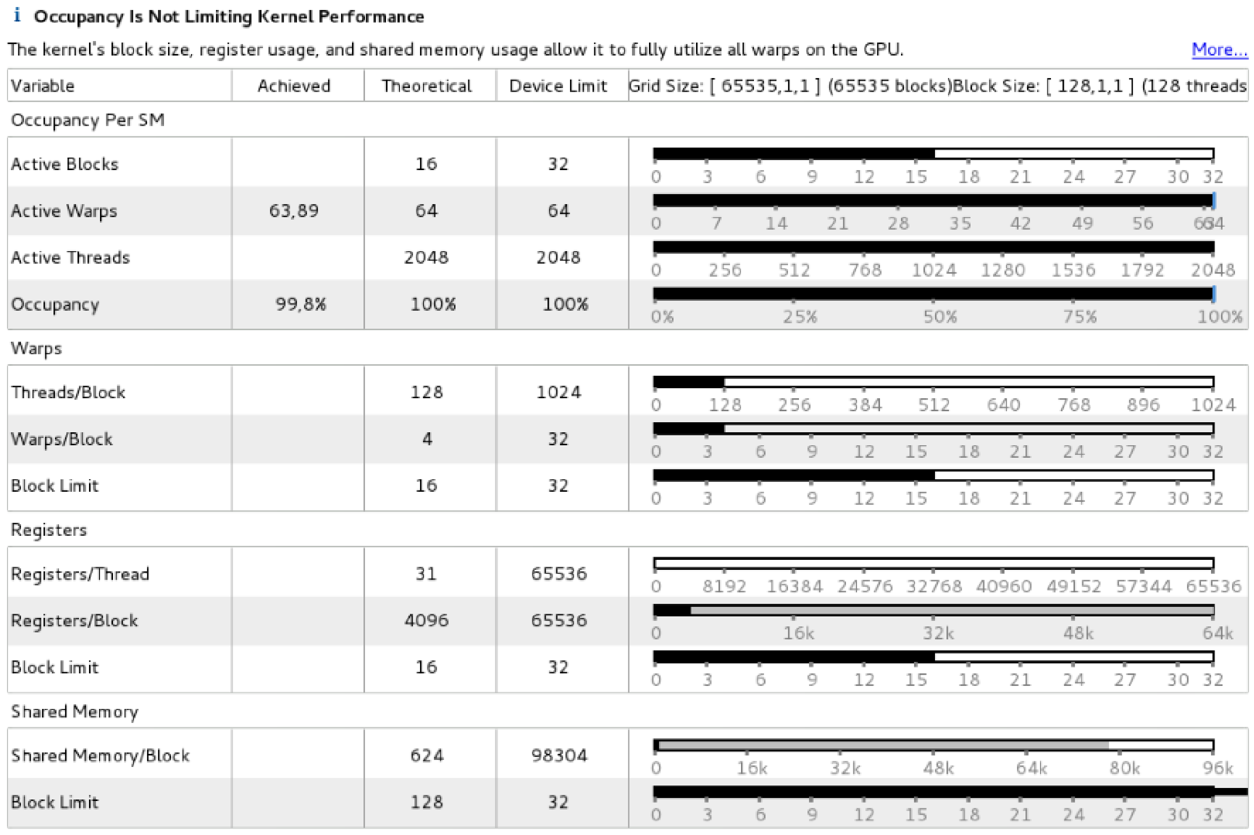


i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1080 Ti". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Figure 17: Kernel performance and profile pie chart results shown by the **nvvp** tool.

The compute units are 72% utilized, while the memory units are around 35% utilized. The memory resource with the higher utilization percentage is the L2 Cache memory.



i Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [65535,1,1] (65535 blocks)Block Size: [128,1,1] (128 threads
Occupancy Per SM				
Active Blocks		16	32	0 3 6 9 12 15 18 21 24 27 30 32
Active Warps	63,89	64	64	0 7 14 21 28 35 42 49 56 6364
Active Threads		2048	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	99,8%	100%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		128	1024	0 128 256 384 512 640 768 896 1024
Warps/Block		4	32	0 3 6 9 12 15 18 21 24 27 30 32
Block Limit		16	32	0 3 6 9 12 15 18 21 24 27 30 32
Registers				
Registers/Thread		31	65536	0 8192 16384 24576 32768 40960 49152 57344 65536
Registers/Block		4096	65536	0 16k 32k 48k 64k
Block Limit		16	32	0 3 6 9 12 15 18 21 24 27 30 32
Shared Memory				
Shared Memory/Block		624	98304	0 16k 32k 48k 64k 80k 96k
Block Limit		128	32	0 3 6 9 12 15 18 21 24 27 30 32

Figure 18: Varying block size, register count and shared memory usage shown by the **nvvp** tool.

The utilization of both resources is low (both compute and memory resources). As a result, it can be assumed that the performance is bounded by latency, and specifically, as introduced by the profiler, instruction and memory latency. Instruction and memory latency limitation indicates that the GPU does not have enough work because instruction execution is stalling excessively.

The pie chart in Figure 17 shows that the three primary latency issues in this kernel are synchronization (35.56%), memory dependency (13.43%) and other (20.59%), which counts for a rank of 69.58 (69.58%) in the optimization importance.

Additionally, Figure 18 shows similar result that almost full parallelism running on the GPU have been achieved. Again, this have been confirmed by the NVIDIA Visual Profiler that occupancy is not limiting the kernel performance due to the block size, register usage and shared memory, which is fully utilizing all warps on the GPU. The achieved Occupancy is 99.8%, while the Theoretical is 100%.

8.6 Application Small vs Large Comparison

After observing the first and final execution, it is essential to compare small vs large performance activity.

8.6.1 Small vs Large 'K' Cluster Problems

Let us now compare the performance of the selected "K" Clusters experimentations where K = 2 & 4 for small problems and K = 500 & 1000 for large problems.

Figure 19 shows the data visualization of performance of selected "K Clusters:

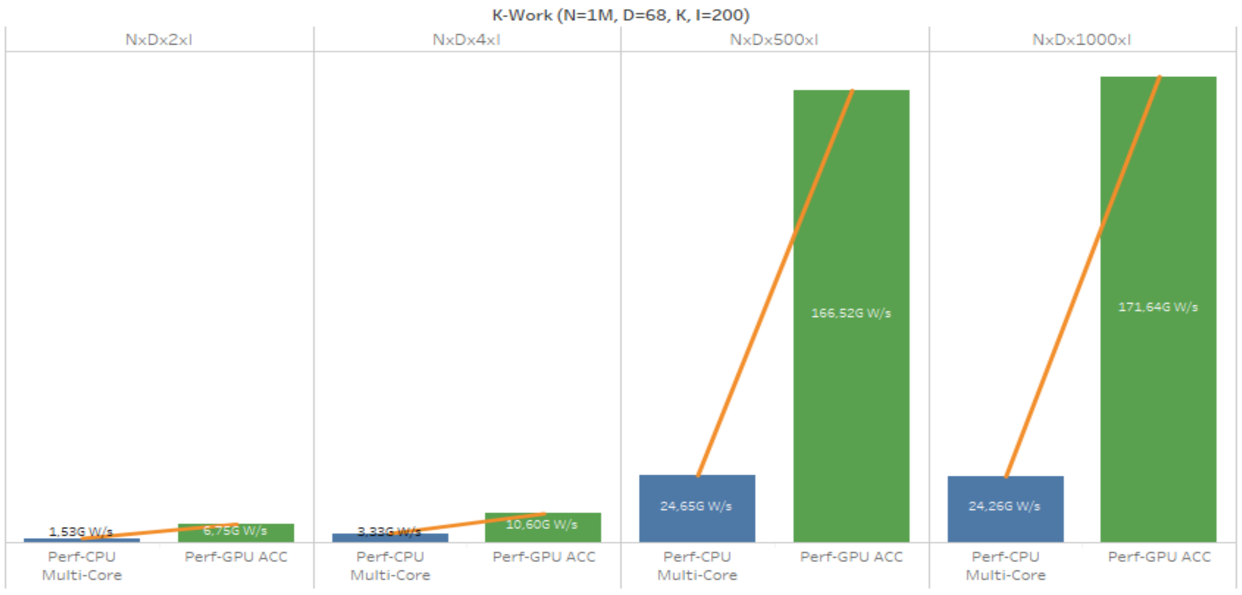


Figure 19: Application Small (K = 2 & 4) vs Large (K = 500 & 1000) performance

8.6.2 Small vs Large ‘D’ Multidimensional Problems

Now, we compare the performance of the selected ‘D’ multidimensional experimentations where D = 2 & 4 for small problems and D = 32 & 68 for large problems.

Figure 20 shows the data visualization of performance of selected ‘D’ dimensions:

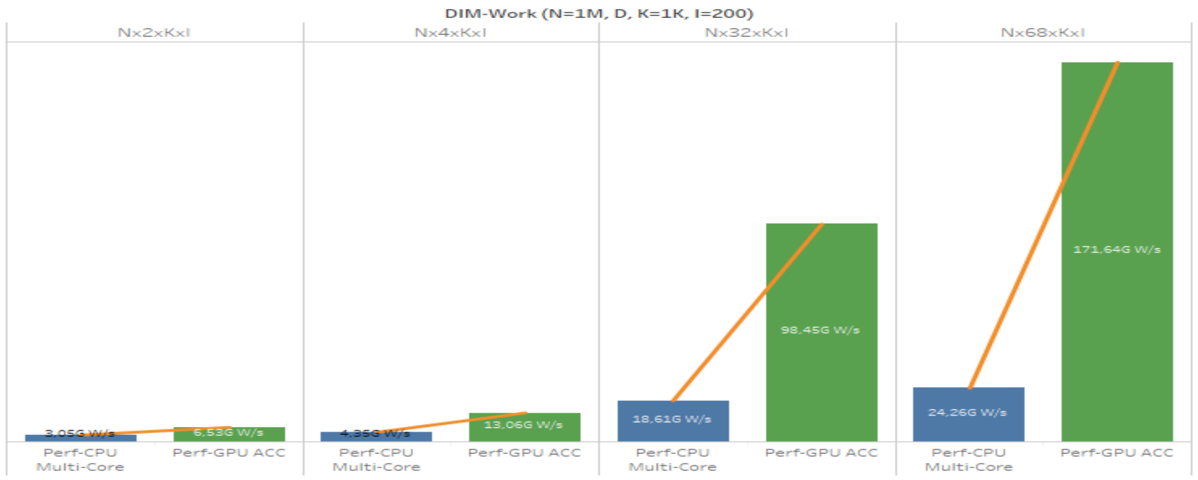


Figure 20: Application Small (D = 2 & 4) vs Large (D = 32 & 68) performance

8.7 Application ‘K’ Clusters and ‘D’ multidimensional Speed-Up

The application activity comparison study now motivates us to observe the speedup of both Multi-Core CPUs and GPU processing the same problem, which can be computed by using the formula:

$$\frac{\boldsymbol{Perf - GPU\_ACC}}{\boldsymbol{Perf - CPU\_Multi\_Core}} = \frac{\boldsymbol{Execution\_Time\ (s) - CPU\_Multi\_Core}}{\boldsymbol{Execution\_Time(s) - GPU\_ACC}}$$

8.7.1 ‘K’ Cluster Speed-Up

Figure 21 shows the data visualization of application speedup of selected “K” Clusters.

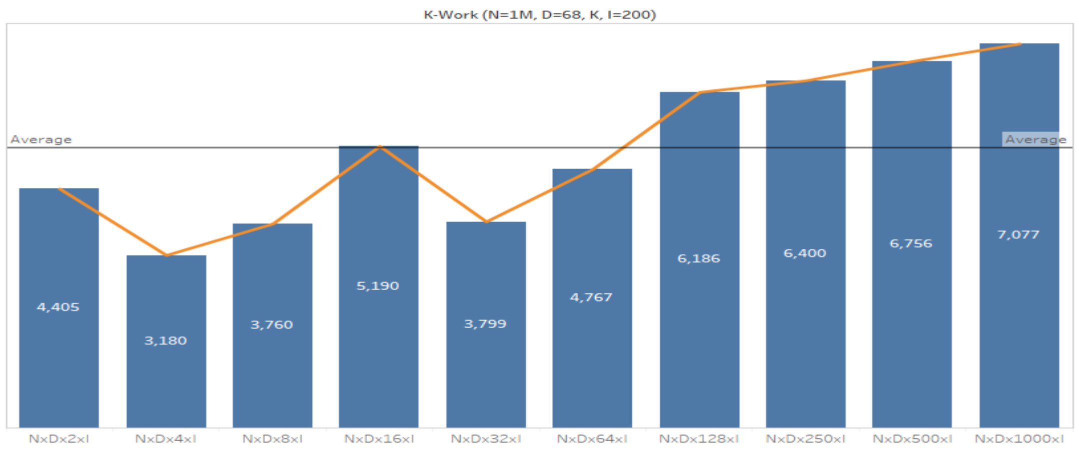


Figure 21: Application Speed-Up of selected “K Clusters (with N = 1M, D = 68, K and I = 200)

8.7.2 ‘D’ Multidimensional Speed-Up

Figure 22 shows the data visualization of application speedup of selected ‘D’ multidimensional.

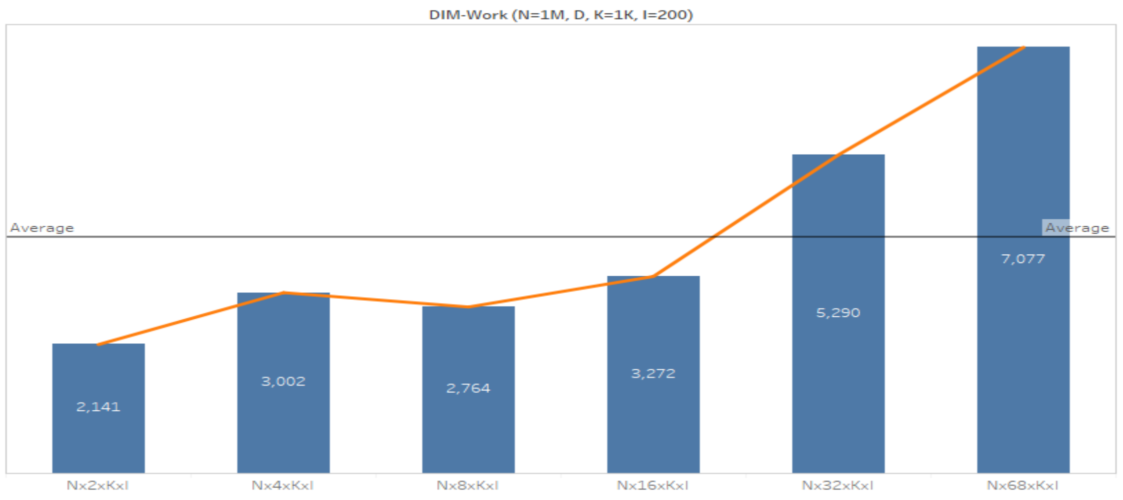


Figure 22: Application Speed-Up of ‘D’ multidimensional (with N = 1M, D, K = 1000 and I = 200)

After observing Figure 19 to 22, the functionality, speedup and stability of the application does extremely well with very large problems. However, on average towards smaller problems, there seems to be an unknown systematic architectural or processing issues occurring within 40%-60% of our experimentations from first to final execution; indicating that our accelerated application should be used for big data problems.

9. APPLICATION INSIGHT

9.1 User Time Savings

As mentioned before, Advanced Analytics goes further and deeper by using more complex and technical methods to accumulate insight that otherwise were unable to detect normally.

Computationally challenging problem with big data, like investigating the effects of increasing “K” Clusters and/or increasing the “D” dimensions of a dataset, depending on the application used, application architectural modifications and/or accelerated techniques applied, have a tremendous tendency to become time overwhelming as the large problem to solve increases in magnitude.

With that said, we will provide application insight to the following exclusive three (3) questions:
1) By increasing the “K” Clusters and/or “D” dimensions of our dataset, how efficient is it timewise?
2) What can we understand from the execution time of the last three (3) large problem experimentations?
3) What will happen to our time usage if we change from the Multi-Core CPUs to the GPU?
Considering all three (3) questions, they all are seeking knowledge on “User Time Savings”.
Let us now investigate the large problems of “K” Clusters and “D” multidimensional user time savings.

9.1.1 "K" Cluster User Time Savings

Figure 23 shows the data visualization of User Time Savings of selected "K" Clusters

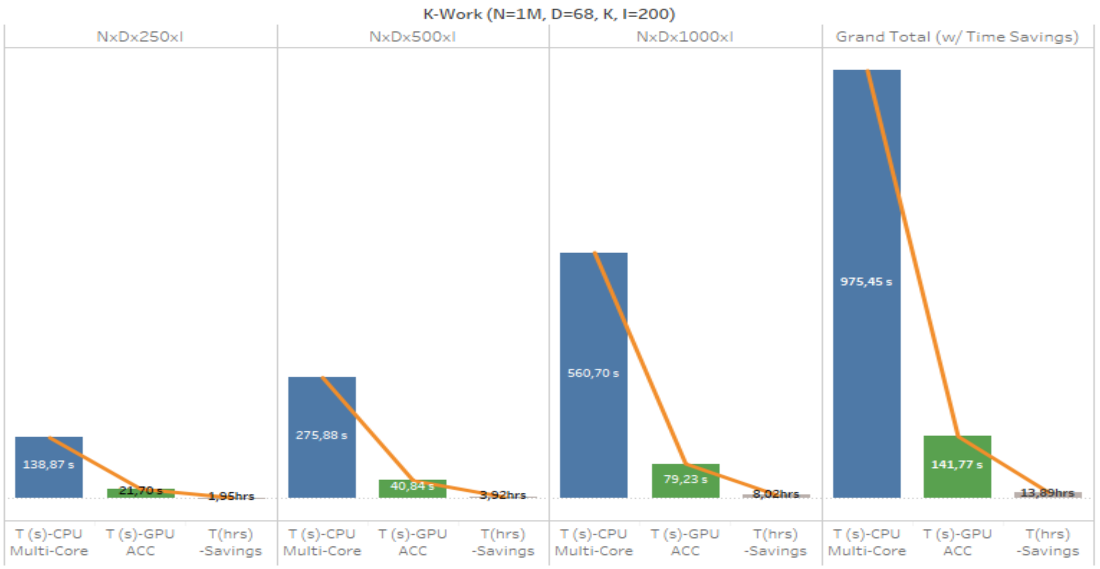


Figure 23: Application User Time Savings of "K" Clusters (K = 250, 500 and 1000)

9.1.2 "D" Multidimensional User Time Savings

Figure 24 shows the data visualization of User Time Savings of selected 'D' multidimensional.

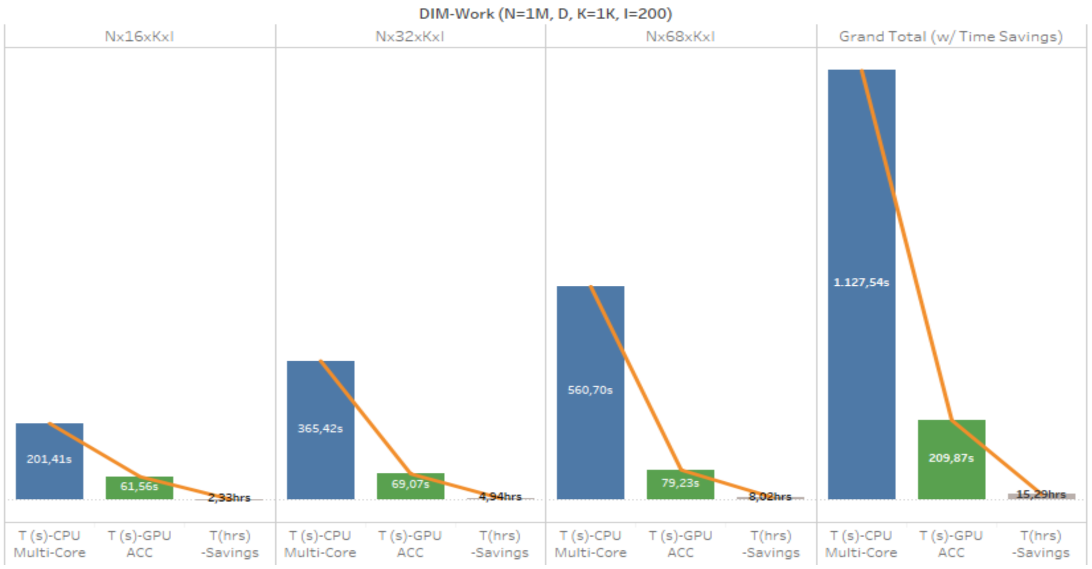


Figure 24: Application User Time Savings of 'D' multidimensional (D = 16, 32 and 68)

In Figure 23 & 24, it can be seen that "D" multidimensional increases (via input datasets) seems to have a higher time saving results verses "K" Cluster increases (via application variable/parameter program modifications). Indicating that we should put more focus on improving the input dataset "D" multidimensional increases or get a bigger dataset with more dimensions, if we seek to understand the futuristic pathway of achieving higher time saving or faster speed from our accelerated application.

From our last three (3) problem experiments, for both cases, the application user time savings of “K” Clusters (K = 250, 500 and 1000) accumulated a total time saving of **13.09 hours** of execution time. And “D” dimensions (D = 16, 32 and 68) increases, accumulating a total time savings of **15.25 hours** of execution time.

Additionally, for large problems (via last three (3) experimentations) changing from using the Multi-Core CPUs to GPU, it is however noticeable, that we would save a total of **28.34 hours** of execution time, which provide extra time of more than a full day of new experimental tasks, developments and/or application insights.

Focusing on our exclusive three (3) questions, this kind of insight, would not have been achieved without the characterizations of Accelerated Applications, Data Mining and GPU Computing; using complex and technical methods to help understand the benefits of our GPU-accelerated implementation of a classical algorithm for unsupervised machine learning.

10. CONCLUSION

We have investigated and developed the principles of Accelerators, GPU Computing and Data Mining within the scientific arena. These principles led to approaches that solve the assumptions of our GPU-accelerated implementation.

The critical step in this research and in **[1],** was to develop an application(s), select a computationally expensive algorithm applied to a dataset(s), utilize standard(s) for parallel computing, simplify the parallel computing process and investigate different approaches of a heterogeneous system. Nevertheless, we have used a different type of programming standard for parallel computing that helped us to provide a fast application performance and significantly reduced our programming efforts for application insight. We have seen this process in our three (3) application design approaches and User Time Savings, particularly in our GPU – Accelerated Version, where we inserted a simple set of OpenACC compiler directives that helped to generate performance improvements and accelerate computation in our application for unknown insight.

The significant task, in this research, was to measure the application performance and study the results of the Multi-Core (MCore) CPUs and GPU accelerated implementations (via OpenACC directives) focusing on application execution, utilization of the GPU device, scalability and speedup. And to understand the importance of small vs large problems of our accelerated application for big data problems; with an official real dataset that was extracted using a Data Extraction System.

Looking at the results of the GPU activity and kernel "PointsToCentrodies_121_gpu", it was discovered that due to the size of our dataset used, selected altered variables ("K" and "D"), small workload size, instruction and memory latency, we were not utilizing the full capacity of the GPU. This was discovered in our first execution that focused on small data problems, the compute units were 22% utilized with the memory units around 5% utilized. And in our final execution that focused on large data problems, the compute units were 72% utilized with the memory units are around 35% utilized. However, in both cases, the memory resources had a higher utilization percentage in the L2 Cache. This was revealed by studying the kernel performance with an achieved consumption on average of 99% of the time. And a significant reason for instruction and memory latency issues were the size of the input dataset(s) and/or data problem(s) investigated.

Noticeably, to have good performance and higher GPU memory usage, with the help of CPU, it is recommended to have a good balance between the selection variable (in the application default settings) and a very large dataset (i.e. BIG DATA). Notwithstanding this limitation, the prospective exploration should be focused on the level of parameters needed to execute any implementation of the k-means clustering algorithm that can make a tremendous difference achieving positive performance analysis. In this research, these parameters are “K”: number of groups, “D”: dimension of points, “N”: number of points and a cautious selection of the “I”: number of iterations; which can be very computationally expensive.

The Scalability and User Time Savings data visualizations shows noticeable advanced analytics insights with a GPU being completely faster than Multi-Cores CPUs, as anticipated when the problem becomes very larger. However, due to the GPU’s low computational power via operations, it is completely useful to incorporate a team effort with the CPU; since the CPU can perform better with many operational tasks. This is to ensure that we have an efficient heterogeneous system with both running correctly. And improve the user experience to resourceful advanced analytics insights with the help of the accumulation and team effort of the GPU and CPU so, all business users and data specialists can make effective proficient resolutions.

We can investigate a futuristic computer architectural approach, with the use of Multi-GPUs devices. However, if all conditions and resources are the same, we may not have a drastic improvement unless we focus on bigger problems that ask questions like “What’s Bigger Than Big Data Problems?” that involves and focuses on finding “Bigger than BIG DATA Problem” insights.

BIBLIOGRAPHY

[1] Janki Bhimani, Miriam Leeser and Ningfang Mi, "Accelerating K- Means clustering with parallel implementations and GPU computing", Proceeding of the High Performance Extreme Computing Conference (HPEC) IEEE, pp. 1-6, 2015.

[2] Mahmoud Al-Ayyoub, Qussai Yaseen, Moahmmed A. Shehab, Yaser Jararweh, Firas Albalas, Elhadj Benkhelifa, "Exploiting GPUs to accelerate clustering algorithms", Computer Systems and Applications (AICCSA) 2016 IEEE/ACS 13th International Conference of, pp. 1-6, 2016.

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

[4] R. Wu, B. Zhang, M. Hsu, "Clustering billions of data points using GPUs", Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop, pp. 1-6, 2009.

[5] NVIDIA Corporation, High Performance Computing, https://developer.nvidia.com/computeworks

[6] Wikipedia contributors. "Machine learning." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 29 May. 2019. Retrieved Web. 2 Jun. 2019.

[7] Crew, Henry, "The Principles of Mechanics". BiblioBazaar, LLC. p. 43. ISBN 978-0-559-36871-4. (2008).

[8] Sebastian Schaetz, Dirk Voit, Jens Frahm, and Martin Uecker, "Accelerated Computing in Magnetic Resonance Imaging: Real-Time Imaging Using Nonlinear Inverse Reconstruction," Computational and Mathematical Methods in Medicine, vol. 2017, Article ID 3527269, 11 pages, 2017. https://doi.org/10.1155/2017/3527269.

[9] Jiawei Han, Micheline Kamber, Data Mining: Concepts and Techniques, London: Academic Press, 5, 2001.

[10] Han, Jiawei, and Micheline Kamber. Data Mining: Concepts and Techniques. San Francisco: Morgan Kaufmann Publishers, 2001. Print.

[11] Fayyad, Usama, Gregory Piatetsky-Shapiro, Padhraic Smyth, From Data Mining to Knowledge Discovery in Databases, 1996.

[12] Nelofar Rehman, "Data Mining Techniques Methods Algorithms and Tools", International Journal of Computer Science and Mobile Computing, Vol.6 Issue.7, July- 2017, pg. 227-231

[13] F.Robert A. Hopgood, Roger J. Hubbold, David A. Duce, eds. Advances in Computer Graphics II. Springer. p. 169. ISBN 9783540169109. "Perhaps the best known one is the NEC 7220", 1986.

[14] "**NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256".** Nvidia. 31 August 1999. Archived from the original on 12 April 2016. Retrieved 28 March 2016, from https://www.nvidia.com/object/IO_20020111_5424.html

[15] Wikipedia contributors. "Graphics processing unit." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 2 Jun. 2019. Retrieved Web. 2 Jun. 2019.

[16] Benson Tao, Vivante Corp. "Understand the mobile graphics processing unit". OpenSystems Media**.** Archived on September 16, 2014. Retrieved 16:11, March 27, 2019 from http://www.embedded-computing.com/embedded-computing-design/understand-the-mobile-graphics-processing-unit

[17] Ma, Zhihua & Wang, Hong & Pu, S.H., “GPU computing of compressible flow problems by a meshless method with space-filling curves”. Journal of Computational Physics. 263. 113-135. 10.1016/j.jcp.2014.01.023. 2014.

[18] Jewell D. Performance Engineering and Management Method — A Holistic Approach to Performance Engineering. In: Liu Z., Xia C.H. (eds) Performance Modeling and Engineering. Springer, Boston, MA. 2008.

[19] C.U. Smith and L.G. Williams. Software Performance Engineering for Object-Oriented Systems: A Use Case Approach. Technical Report, Performance Engineering Services, 1998.

[20] Han, Jiawei, and Micheline Kamber. Data Mining: Concepts and Techniques, Third Edition. 3rd ed. Waltham, Mass.: Morgan Kaufmann Publishers, 2012.

[21] Singh, Archana K., Avantika Yadav and Ajay Rana. “K-means with Three different Distance Metrics.” 2013.

[22] Komal D. Nistane1 and Shailendra W. Shende. GPU Accelerated Clustering Techniques. International Journal of Science and Research. Volume 4 Issue 4, April 2015.

[23] Tutorials Point. Performance Testing. https://www.tutorialspoint.com/software_testing_dictionary/performance_testing.htm

[24] Wikipedia contributors. "Performance engineering." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 17 Mar. 2019. Retrieved Web. 6 Apr. 2019.

[25] cplusplus.com. Reference. http://www.cplusplus.com/reference/

[26] Wikipedia contributors. "Optimizing compiler." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 19 Mar. 2019. Retrieved Web. 19 Apr. 2019.

[27] Wikipedia contributors. "OpenACC." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 7 Jan. 2019. Retrieved Web. 19 Apr. 2019.

[28] OpenACC Directives, “OpenACC: More Science Less Programming.” https://developer.nvidia.com/openacc

[29] Dua, D. and Graff, C. “UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]”. Irvine, CA: University of California, School of Information and Computer Science. 2019.

[30] UCI Machine Learning Repository. US Census Data (1990) Data Set. https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990)

[31] Wikipedia contributors. “Slurm Workload Manager”. Wikipedia, The Free Encyclopedia. March 5, 2019, 00:53 UTC. Retrieved April 19, 2019.

[32] SchedMD®. “Frequently Asked Questions”. Slurm Workload Manager. https://slurm.schedmd.com/faq.html

[33] DataFlair ©. “Tableau Tutorial For Beginners”. https://data-flair.training/blogs/tableau-tutorial/

[34] Wikipedia contributors. “Tableau Software”. Wikipedia, The Free Encyclopedia. March 7, 2019, 08:20 UTC. Retrieved April 21, 2019.

[35] Wikipedia contributors. "Speedup." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 11 Apr. 2019. Retrieved Web. 22 Apr. 2019.

[36] Wikipedia contributors. "Heat map." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 26 Apr. 2019. Retrieved Web. 27 Apr. 2019.

A. APPENDIX

A.1 Baseline Version 1 (Random Dataset)

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>


// update the center for all the centroids.
// return:
// 0 if the centroids do not change,
// 1 otherwise
int recalcularCentre(int *Punts, int N, int D, int K, double *Centroides, int
*PC, int *Sep)
{
      int i, j, c, x;
      double* v = (double*)malloc(sizeof(double)*D);         //Auxiliary
vector to calculate the new position of the centroids
      int r = 0;

      for(i=0; i<K; i++){
            // initialize V to 0;
            for(x=0; x<D; x++)
                  v[x] = 0;

            // update the centre of i-th centroid
            for(j=0; j<Sep[i]; j++){
                  for(c=0; c<D; c++)
                        v[c] += (double)Punts[PC[i*N+j]*D+c];
            }
            for(x=0; x<D; x++){
                  v[x] = v[x] / (double)Sep[i];

                  // if centre is updated, set flag <r> to 1.
                  if(v[x] != Centroides[(i*D)+x])
                        r = 1;
                  // update with new centroid
                  Centroides[i*D+x] = v[x];
            }
      }
      free(v);
      return r;
}

//It generates N points with D - dimensions randomly and makes the choice of
centroid
void generatePunts(int *Punts, int N, int D, int K, double *Centroides, int
DIMENSION)
{
      int i, j;
      // generate N points(D-th vector) randomly
      for(i=0; i<N; i++){
            for(j=0; j<D; j++){
                Punts[(D*i)+j] = rand() % DIMENSION;
            }
        }

        printf("\nGenerated %d x %d random data.\n", N, D);

        // assign K points as center of each centroids
        for(i=0; i<K; i++){
            int c = rand() % N;
            for(j=0; j<D; j++){
                Centroides[(i*D)+j] = (double)Punts[(c*D)+j];
            }
        }

        printf("\nCentroides:\n\t");
        for(i=0; i<C; i++){
            for(j=0; j<D; j++)
                printf("%3.1f\t",Centroides[i*D+j]);
            printf("\n\t");
        }
}

//Returns the position(centroid number) where the minimum value of a vector
is found
int mmin(double* v, int K)
{
        double m = v[0];
        int i, r = 0;
        for (i = 1; i<K; i++) {
            if (m > v[i]) {
                r = i;
                m = v[i];
            }
        }
        return r;
}

// Assign each point to the nearest centroid
void PointsToCentroides(int *Punts, int N, int D, int K, double *Centroides,
int *PC, int *Sep){
        int i, j, c, m;
        double* dist = (double*)malloc(sizeof(double)*K);
        for(i=0; i<K; i++)
            Sep[i]=0;           // Number of assigned points in each class
        for(i=0; i<N; i++){

            for(j=0; j<K; j++){
                dist[j] = 0;
                for(c=0; c<D; c++){
                //Let's calculate the distance from point i to centered j
                    dist[j] += ((double)Punts[i*D+c] - Centroides[j*D+c])
* ((double)Punts[i*D+c] - Centroides[j*D+c]);
                }
            }
            //We take the minimum distance between a point and all centroids
and assign it to the nearest centroide
            m = mmin(dist, K);

            // centroid (m)  <--  point (i)
            PC[m*N+Sep[m]] = i;
            Sep[m]+=1;
      }
      free(dist);
}


void main(int argc, char **argv){

      int N;                  //Number of points
      int D;                  //Dimensions of the point
      int K;                  //Number of centroides
      int DIMENSION;    //Dimension of the point space
      srand(7);

      if(argc < 5){
            // Default parameters
            N = 1000;
            D = 3;
            K = 4;
            DIMENSION = 1000;
      }
      else{
            // Parameters passed by argument
            N = atoi(argv[1]);
            D = atoi(argv[2]);
            K = atoi(argv[3]);
            DIMENSION = atoi(argv[4]);
      }

      int *Punts = (int*) malloc( sizeof(int)*N*D );
      //Points matrix
      double *Centroides = (double*) malloc( sizeof(double)*K*D );
      //Centroides
      int *PC = (int*) malloc( sizeof(int)*N*K );
      //Index array of points which assigned to each centroid
      int *Sep = (int*) malloc( sizeof(int)*K );
      //Number of points for each centroid
      int cont = 0;
                  // iteration count
      int final;
                  // flag of convergence

      printf("K-means:\n\t Points: %d\n\t Dimension: %d\n\t Centroides:
%d\n",N,D,K);
      printf("Points space is from %d x %d\n", DIMENSION, D);

      // initial generation of N x D points
      generatePunts(Punts, N, D, K, Centroides, DIMENSION);
```

```
	do{
		// reassignment each points to nearest centroids
		PointsToCentroides(Punts, N, D, K, Centroides, PC, Sep);

		// counting iteration round
		cont += 1;

		// check if no changed
		final = recalcularCentre(Punts, N, D, K, Centroides, PC, Sep);
	}while(final && cont<200);

	printf("\nNumber of iterations needed: %d\n", cont);

	printf("\nNew centroides:\n\t");
	int i = 0, j = 0;
	for (i = 0; i<K; i++) {
		for (j = 0; j<D; j++)
			printf("%3.1f\t", Centroides[i*D + j]);
		printf("\n\t");
	}
	printf("\n");
}
```

A.2 Baseline Version 2 (Input Dataset)

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <limits.h>

// update the center for all the centroids.
// return:
// 0 if the centroids do not change,
// 1 otherwise
int recalcularCentre(float* Punts, int N, int D, int K, float* Centroides,
int* PC, int *Sep)
{
	int i, j, c, x;
	float* v = (float*)malloc(sizeof(float) * D); //Auxiliary vector to
calculate the new position of the centroids
	int r = 0;

	for(i=0; i<K; i++){
		// initialize V to 0;
		for(x=0; x<D; x++)
			v[x] = 0;

		// update the centre of i-th centroid
		for(j=0; j<Sep[i]; j++){
			for(c=0; c<D; c++)
				v[c] += (float)Punts[PC[i*N+j]*D+c];
		}
		for(x=0; x<D; x++){
			v[x] = v[x] / (float)Sep[i];
```

```
                        // if centre is updated, set flag <r> to 1.
                        if(v[x] != Centroides[(i*D)+x])
                              r = 1;
                        // update with new centroid
                        Centroides[i*D+x] = v[x];
                  }
            }
            free(v);
            return r;
}

//It generates N points with D - dimensions randomly and makes the choice of
centroid
int read_CSV_data(char* filename, char**row_name, float* Punts, int N, int D,
int K, float* Centroides)
{
      // open input csv data file
      FILE* fp = fopen(filename, "r");
      // if failed to open file, exit print error message and program
      if (!fp) {
            printf("Can't find input file - %s\n", filename);
            return 1;
      }

      printf("reading input data from %s\n", filename);

      int i=0, j=0, n_read = 0;                 // n_read: number of acually
read lines
      int n_total_idx = 0;                // unique index of reading float
data
      char line[1024];                          // one line to be read from
csv file

      // read first line (caseid,....)
      fgets(line, sizeof(line), fp);

      // loop reading line by line from input file
      while (fgets(line, sizeof(line), fp)) {
            // get first string in line with separating comma
            char* token = strtok(line, ",");
            j = 0;
            int n_each_in_line = 0;       // number of float values in each
line

            // get all float value from line
            while (token != NULL) {
                  // if arrived to end of line, break
                  if (strcmp(token, "\n") == 0)       // compare two strings
- token and "\n" and if equals
                        break;
                  if (j == 0) {           // first string at line (means row
name)
                        strcpy(row_name[i], token);
                        j += 1;
                        token = strtok(NULL, ",");
                        continue;
                    }
// convert string to float value and is sent to Punts array
                    Punts[n_total_idx++] = atof(token);
                    n_each_in_line += 1; // increase number of read float value
                    j += 1;
                    // if each line includes more values than <D>, ignore 
remain values
                    if(n_each_in_line >= D)
                        break;
                    // get next string from line
                    token = strtok(NULL, ",");
                }

                // if number of values at each line doesnot equal to <D>, print 
error message and exit
                if (n_each_in_line != D) {
                    printf("Now each line components is %d\n", n_each_in_line);
                    printf("\nEach line should include %d values.\n", D);
                    return 2;
                }

                // increase number of read line
                i += 1;
                n_read += 1;
                // if line number is greater than <N>, ignore remain lines
                if (n_read > N)
                    break;
        }

        // close opened file
        fclose(fp);

        // confirm read-lines is equal to <N>
        assert(n_read == N);

        // if n_read is less than <N>, set <N> as n_read.
        N = n_read;
        printf("\nRead %d x %d points data from %s.\n", N, D, filename);

        // assign K points as center of each centroids
        for (i = 0; i<K; i++) {
                int c = rand() % N;
                for (j = 0; j<D; j++) {
                        Centroides[(i*D) + j] = Punts[(c*D) + j];
                }
        }

        printf("\nCentroides:\n\t");
        for (i = 0; i<K; i++) {
                for (j = 0; j<D; j++)
                        printf("%3.1f\t", Centroides[i*D + j]);
                printf("\n\t");
        }
```

```
	return 0;
}

//Returns the position(centroid number) where the minimum value of a vector
is found
int mmin(float* v, int K)
{
	float m = v[0];
	int i, r = 0;
	for (i = 1; i<K; i++) {
		if (m > v[i]) {
			r = i;
			m = v[i];
		}
	}
	return r;
}

// Assign each point to the nearest centroid
void PointsToCentroides(float *Punts, int N, int D, int K, float *Centroides,
int *PC, int *Sep){
	int i, j, c, m;
	float* dist = (float*)malloc(sizeof(float)*K);
	for(i=0; i<K; i++)
		Sep[i]=0;		// Number of assigned points in each class
	for(i=0; i<N; i++){

		for(j=0; j<K; j++){
			dist[j] = 0;
			for(c=0; c<D; c++){
			//Let's calculate the distance from point i to centered j
				dist[j] += ((float)Punts[i*D+c] - Centroides[j*D+c])
* ((float)Punts[i*D+c] - Centroides[j*D+c]);
			}
		}
		//We take the minimum distance between a point and all centroids
and assign it to the nearest centroide
		m = mmin(dist, K);

		// centroid (m)  <--  point (i)
		PC[m*N+Sep[m]] = i;
		Sep[m]+=1;
	}
	free(dist);
}

int main(int argc, char **argv){

	int N;			//Number of points
	int D;			//Dimensions of the point
	int K;			//Number of centroides

	// string array to be read file name
	char* file = (char*)malloc(1024);
	// if all parameters don't given, set as default parameter
	if (argc < 5) {
		// Default parameters
		strcpy(file, "InputData_Large.csv");
		K = 4;
		N = 1000000;
		D = 68;

		printf("Usage: %s<InputData_Large.csv>, <K=Number of Clusters>,
<N=Number of Rows>, <D=Number of Columns>\n", argv[0]);
		printf("We now use default parameters with InputData_Large.csv,
K=%d, N=%d, D=%d\n",K,N,D);
	}
	else {
		// copy file name from argument to variable
		strcpy(file, argv[1]);
		K = atoi(argv[2]);
		N = atoi(argv[3]);
		D = atoi(argv[4]);
	}

	// allocate memory for points matrix, centres and so on
	float *Punts = (float*)malloc(sizeof(float) * N * D);
	//Points matrix
	float *Centroides = (float*) malloc(sizeof(float) * K * D );
	//Centroides
	int *PC = (int*) malloc( sizeof(int)*N*K );
	//Index array of points which assigned to each centroid
	int *Sep = (int*) malloc( sizeof(int)*K );
	//Number of points for each centroid
	int cont = 0;			// iteration count
	int final;			// flag of convergence
	int c;

	// allocate string array for caseid (row name)
	char** row_name = (char**)malloc(sizeof(char*) * N);
	for (c = 0; c < N; c++)
		row_name[c] = (char*)malloc(sizeof(char) * 10);

	printf("K-means:\n\t Points: %d\n\t Dimension: %d\n\t Centroides:
%d\n",N,D,K);

	// read input data from csv file
	if(read_CSV_data(file, row_name, Punts, N, D, K, Centroides) > 0)
		exit(1);

	do{
		// reassignment each points to nearest centroids
		PointsToCentroides(Punts, N, D, K, Centroides, PC, Sep);

		// counting iteration round
		cont += 1;

		// check if no changed
		final = recalcularCentre(Punts, N, D, K, Centroides, PC, Sep);
```

```c
}while(final && cont<200);

printf("\nNumber of iterations needed (T): %d\n", cont);
printf("\nNumber of changes (F): %d\n", final);

// exit(0);

printf("\nNew centroides:\n\t");
int i = 0, j = 0;
for (i = 0; i<K; i++) {
      for (j = 0; j<D; j++)
            printf("%3.1f\t", Centroides[i*D + j]);
      printf("\n\t");
}
printf("\n");

// get yellow line
char line[1024];         // one line to be read from csv file
FILE* fp = fopen(file, "r");  // read first line (caseid,....)
fgets(line, sizeof(line), fp);
fclose(fp);

// print new clusters to result csv file
FILE* result_file = fopen("result.csv", "w");
if (!result_file) {
      printf("Can't create result file.\n");
      return 1;
}
fprintf(result_file, "Number of clusters: %d\n", K);
for (c = 0; c < K; c++) {
fprintf(result_file, "\tCluster-%d : %d elements\n", c + 1, Sep[c]);
}

for (i = 0; i < K; i++) {
      fprintf(result_file, "\nCluster - %d\n", i + 1);
      fprintf(result_file, "\t%s\n", line);

      for (j = 0; j < Sep[i]; j++) {
            fprintf(result_file, "\t%s,", row_name[PC[i*N + j]]);
            for (c = 0; c < D; c++) {
            fprintf(result_file, "%3.1f,", Punts[PC[i*N+j]*D+c]);
            }
            fprintf(result_file, "\n");
      }
}
// close file
fclose(result_file);

// free allocated memory
free(file);
free(Punts);
free(Centroides);
free(PC);
free(Sep);
free(row_name);
```

```
      exit(0);
}
```

A.3 Optimized Version

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>

//It generates N points with D - dimensions randomly and makes the choice of
centroid
int read_CSV_data(char* filename, char**row_name, float* Punts, int N, int D,
int K, float* Centroides)
{
      // open input csv data file
      FILE* fp = fopen(filename, "r");
      // if failed to open file, exit print error message and program
      if (!fp) {
            printf("Can't find input file - %s\n", filename);
            return 1;
      }

      printf("reading input data from %s\n", filename);

      int i=0, j=0, n_read = 0;      // n_read: number of acually read lines
      int n_total_idx = 0;     // unique index of reading float data
      char line[1024];               // one line to be read from csv file

      // read first line (caseid,....)
      fgets(line, sizeof(line), fp);

      // loop reading line by line from input file
      while (fgets(line, sizeof(line), fp)) {
            // get first string in line with separating comma
            char* token = strtok(line, ",");
            j = 0;
            int n_each_in_line = 0; // number of float values in each line

            // get all float value from line
            while (token != NULL) {
                  // if arrived to end of line, break
                  if (strcmp(token, "\n") == 0)       // compare two strings
- token and "\n" and if equals
                        break;
                  if (j == 0) {      // first string at line (means row name)
                        strcpy(row_name[i], token);
                        j += 1;
                        token = strtok(NULL, ",");
                        continue;
                  }
                  Punts[n_total_idx++] = atof(token); // convert string to
float value and is sent to Punts array
                  n_each_in_line += 1; // increase number of read float value
                  j += 1;
      // if each line includes more values than <D>, ignore remain values
                  if(n_each_in_line >= D)
                        break;
                  // get next string from line
                  token = strtok(NULL, ",");
            }

            // if number of values at each line doesnot equal to <D>, print
error message and exit
            if (n_each_in_line != D) {
                  printf("Now each line components is %d\n", n_each_in_line);
                  printf("\nEach line should include %d values.\n", D);
                  return 2;
            }

            // increase number of read line
            i += 1;
            n_read += 1;
            // if line number is greater than <N>, ignore remain lines
            if (n_read > N)
                  break;
      }

      // close opened file
      fclose(fp);

      // confirm read-lines is equal to <N>
      assert(n_read == N);

      // if n_read is less than <N>, set <N> as n_read.
      N = n_read;
      printf("\nRead %d x %d points data from %s.\n", N, D, filename);

      // assign C points as center of each centroids
      for (i = 0; i<K; i++) {
            int c = rand() % N;
            for (j = 0; j<D; j++) {
                  Centroides[(i*D) + j] = Punts[(c*D) + j];
            }
      }

      printf("\nCentroides:\n\t");
      for (i = 0; i<K; i++) {
            for (j = 0; j<D; j++)
      printf("%3.1f\t", Centroides[i*D + j]);
      printf("\n\t");
      }

      return 0;
}
```

```
// Assign each point to the nearest centroid
// Update the center for all the centroid

int PointsToCentroides(float *Punts, int N, int D, int K, float *Centroides,
float *New_Centr, int *Labels, int *Sep)
{
      int i, j, c, min_arg1, min_arg2, r=0;
      float dist1, dist2, dist3, dist4, min_dist1, min_dist2;

        min_dist1= min_dist2= FLT_MAX;

      for(i=0; i<K; i++)
        {
         Sep[i]=0;             // Number of assigned points in each class
           for (c=0; c<D; c++)
           New_Centr[i*D+c] = 0;
        }

      for(i=0; i<N; i+=2)
        {
           // FIXME: only works if N is even
              dist1= dist2= 0; j=0;
            if (K&1 == 1)
                {   // even number
              for(c=0; c<D; c++)
                  {
                      float punt1= Punts[i*D+c];
                      float punt2= Punts[(i+1)*D+c];
                      float centr= Centroides[j*D+c];
                  float temporal1 = punt1 - centr;
                  float temporal2 = punt2 - centr;
                  dist1 += temporal1*temporal1;
                  dist2 += temporal1*temporal1;
              }
                  min_arg1 = (dist1<min_dist1)? j    : min_arg1;
                  min_dist1= (dist1<min_dist1)? dist1: min_dist1;

                  min_arg2 = (dist2<min_dist2)? j    : min_arg2;
                  min_dist2= (dist2<min_dist2)? dist2: min_dist2;

                  j= 1;
                  dist1=dist2=0;
            }

              dist3= dist4= 0;

            for(; j<K; j+=2)
                {
             for(c=0; c<D; c++)
                   {
                      float punt1 = Punts[i*D+c];
                      float punt2 = Punts[(i+1)*D+c];
                      float centr1= Centroides[j*D+c];
                      float centr2= Centroides[(j+1)*D+c];
                 //Let's calculate the distance from point i to centered j
                  float temporal1 = punt1 - centr1;
                  float temporal2 = punt1 - centr2;
                  float temporal3 = punt2 - centr1;
                  float temporal4 = punt2 - centr2;

                  dist1 += temporal1*temporal1;
                  dist2 += temporal2*temporal2;
                  dist3 += temporal3*temporal3;
                  dist4 += temporal4*temporal4;
                }

                  min_arg1 = (dist1<min_dist1)? j     : min_arg1;
                  min_dist1= (dist1<min_dist1)? dist1: min_dist1;

                  min_arg2 = (dist3<min_dist2)? j     : min_arg2;
                  min_dist2= (dist3<min_dist2)? dist3: min_dist2;

                  min_arg1 = (dist2<min_dist1)? j+1    : min_arg1;
                  min_dist1= (dist2<min_dist1)? dist1: min_dist1;

                  min_arg2 = (dist4<min_dist2)? j+1    : min_arg2;
                  min_dist2= (dist4<min_dist2)? dist4: min_dist2;
            }

            // centroid (min_arg1)  <--  point (i)
            Labels[i] = min_arg1;
            Sep[min_arg1]+=1;
            for (c=0; c<D; c++)
               New_Centr[min_arg1*D+c] += Punts[i*D+c];

            // centroid (min_arg2)  <--  point (i+1)
            Labels[i+1] = min_arg2;
            Sep[min_arg2]+=1;
            for (c=0; c<D; c++)
               New_Centr[min_arg2*D+c] += Punts[(i+1)*D+c];
      }
        // all points have been processed

      for (j=0; j<K; j++)
        {  // for every cluster

         for (c=0; c<D; c++)
           {
           float value = New_Centr[j*D+c] / (float)Sep[j];

           // if centre is updated, set flag <r> to 1.
           if (value != Centroides[(j*D)+c])
               r = 1;

           // update with new centroid
             Centroides[j*D+c] = value;
           }
         }
   return r;
}


int main(int argc, char **argv){

      int N;                    //Number of points
      int D;                    //Dimensions of the point
      int K;                    //Number of centroides

      // string array to be read file name
      char* file = (char*)malloc(1024);
      // if all parameters don't given, set as default parameter
      if (argc < 5) {
            // Default parameters
            strcpy(file, "InputData_Large.csv");
            K = 4;
            N = 1000000;
            D = 68;

            printf("Usage: %s<InputData_Large.csv>, <K=Number of Clusters>,
<N=Number of Rows>, <D=Number of Columns>\n", argv[0]);
            printf("We now use default parameters with InputData_Large.csv,
K=%d, N=%d, D=%d\n",K,N,D);
      }
      else {
            // copy file name from argument to variable
            strcpy(file, argv[1]);
            K = atoi(argv[2]);
            N = atoi(argv[3]);
            D = atoi(argv[4]);
      }

      // allocate memory for points matrix, centres and so on
      float *Punts      = (float*) malloc( sizeof(float) * N * D );     //
Points matrix
      int   *Labels     = (int*)   malloc( sizeof(int)   * N );   // Centroid
assigned to Point
      float *Centroides = (float*) malloc( sizeof(float) * K * D );     //
Centroides
      float *New_Centr  = (float*) malloc( sizeof(float) * K * D );     //
New centroides
      int   *Sep        = (int*)   malloc( sizeof(int)   * K );       //
Number of points for each centroid
      int cont = 0;     // iteration count
      int final;  // flag of convergence
      int c;

      // allocate string array for caseid (row name)
      char** row_name = (char**)malloc(sizeof(char*) * N);
      for (c = 0; c < N; c++)
            row_name[c] = (char*)malloc(sizeof(char) * 10);

      printf("K-means:\n\t Points: %d\n\t Dimension: %d\n\t Centroides:
%d\n",N,D,K);
```

```
        // read input data from csv file
        if(read_CSV_data(file, row_name, Punts, N, D, K, Centroides) > 0)
              exit(1);

        do {
          // reassignment each points to nearest centroids
          final = PointsToCentroides(Punts, N, D, K, Centroides, New_Centr,
Labels, Sep);
          cont += 1;
        }
          while (final && cont<200);

        printf("\nNumber of iterations needed (T): %d\n", cont);
        printf("\nNumber of changes (F): %d\n", final);

         // exit(0);

        printf("\nNew centroides:\n\t");
        int i = 0, j = 0;
        for (i = 0; i<K; i++) {
              for (j = 0; j<D; j++)
                    printf("%3.1f\t", Centroides[i*D + j]);
              printf("\n\t");
        }
        printf("\n");

        // get yellow line
        char line[1024];        // one line to be read from csv file
        FILE* fp = fopen(file, "r");  // read first line (caseid,....)
        fgets(line, sizeof(line), fp);
        fclose(fp);

        // print new clusters to result csv file
        FILE* result_file = fopen("result.csv", "w");
        if (!result_file) {
              printf("Can't create result file.\n");
              return 1;
        }
        fprintf(result_file, "Number of clusters: %d\n", K);
        for (c = 0; c < K; c++) {
        fprintf(result_file, "\tCluster-%d : %d elements\n", c + 1, Sep[c]);
        }

        for (i = 0; i < K; i++) {
              fprintf(result_file, "\nCluster - %d\n", i + 1);
              fprintf(result_file, "\t%s\n", line);

              for (j = 0; j < N; j++) {
                    if (Labels[i] == i)
                {
                  fprintf(result_file, "\t%s,", row_name[i]);
                    for (c = 0; c < D; c++)
                      {
                    fprintf(result_file, "%3.1f,", Punts[i*D+c]);
                    }
                    fprintf(result_file, "\n");
                }
                    }
        }
        // close file
        fclose(result_file);

        // free allocated memory
        free(file);
        free(Punts);
        free(Centroides);
        free(New_Centr);
        free(Labels);
        free(Sep);
        free(row_name);

        exit(0);
}
```

A.4 Accelerated Version

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>

//It generates N points with D - dimensions randomly and makes the choice of
centroid
int read_CSV_data(char* filename, char**row_name, float* Punts, int N, int D,
int K, float* Centroides)
{
        // open input csv data file
        FILE* fp = fopen(filename, "r");
        // if failed to open file, exit print error message and program
        if (!fp) {
              printf("Can't find input file - %s\n", filename);
              return 1;
        }

        printf("reading input data from %s\n", filename);

        int i=0, j=0, n_read = 0;      // n_read: number of acually read lines
        int n_total_idx = 0;           // unique index of reading float data
        char line[1024];               // one line to be read from csv file

        // read first line (caseid,....)
        fgets(line, sizeof(line), fp);

        // loop reading line by line from input file
        while (fgets(line, sizeof(line), fp)) {
              // get first string in line with separating comma
              char* token = strtok(line, ",");
```

```
            j = 0;
            int n_each_in_line = 0; // number of float values in each line
            // get all float value from line

            while (token != NULL) {
                  // if arrived to end of line, break
                  if (strcmp(token, "\n") == 0)       // compare two strings
- token and "\n" and if equals
                        break;
                  if (j == 0) {     // first string at line (means row name)
                        strcpy(row_name[i], token);
                        j += 1;
                        token = strtok(NULL, ",");
                        continue;
                  }

                  Punts[n_total_idx++] = atof(token); // convert string to
float value and is sent to Punts array
                  n_each_in_line += 1; // increase number of read float value
                  j += 1;
                  // if each line includes more values than <D>, ignore
remain values
                  if(n_each_in_line >= D)
                        break;
                  // get next string from line
                  token = strtok(NULL, ",");
            }

            // if number of values at each line doesnot equal to <D>, print
error message and exit
            if (n_each_in_line != D) {
                  printf("Now each line components is %d\n", n_each_in_line);
                  printf("\nEach line should include %d values.\n", D);
                  return 2;
            }

            // increase number of read line
            i += 1;
            n_read += 1;
            // if line number is greater than <N>, ignore remain lines
            if (n_read > N)
                  break;
      }

      // close opened file
      fclose(fp);

      // confirm read-lines is equal to <N>
      assert(n_read == N);

      // if n_read is less than <N>, set <N> as n_read.
      N = n_read;
      printf("\nRead %d x %d points data from %s.\n", N, D, filename);
      // assign K points as center of each centroids
      for (i = 0; i<K; i++) {
            int c = rand() % N;
            for (j = 0; j<D; j++) {
                  Centroides[(i*D) + j] = Punts[(c*D) + j];
            }
      }

      printf("\nCentroides:\n\t");

      for (i = 0; i<K; i++) {
            for (j = 0; j<D; j++)
                  printf("%3.1f\t", Centroides[i*D + j]);
            printf("\n\t");
      }

      return 0;
}


// Assign each point to the nearest centroid
// Updates the center for all the centroid
int PointsToCentroides(float *Punts, int N, int D, int K, float *Centroides,
float *New_Centr, int *Labels, int *Sep)
{
    int i, r=0;
    int *pR = &r;

  #pragma acc data present(Punts,Labels,Centroides,New_Centr,Sep) copy(pR[1])
    {
      #pragma acc parallel loop gang vector_length(128)
      for (i=0; i<K; i++)
        {
           int c;
         Sep[i]=0;             // Number of assigned points in each class
             #pragma acc loop vector
           for (c=0; c<D; c++)
           New_Centr[i*D+c] = 0;
        }

          #pragma acc parallel loop gang vector_length(128)
        for(i=0; i<N; i+=2)
        {
          // FIXME: only works if N is even

        int j, c, min_arg1, min_arg2;
          float dist1, dist2, dist3, dist4, min_dist1, min_dist2;

          min_dist1= min_dist2= FLT_MAX;

              dist1= dist2= 0; j=0;
            if (K&1 == 1)
                {   // even number

          #pragma acc loop vector
              for(c=0; c<D; c++)
                  {
```

```
                float punt1= Punts[i*D+c];
                float punt2= Punts[(i+1)*D+c];
                float centr= Centroides[j*D+c];
            float temporal1 = punt1 - centr;
            float temporal2 = punt2 - centr;
            dist1 += temporal1*temporal1;
            dist2 += temporal1*temporal1;
        }
            min_arg1 = (dist1<min_dist1)? j     : min_arg1;
            min_dist1= (dist1<min_dist1)? dist1: min_dist1;

            min_arg2 = (dist2<min_dist2)? j     : min_arg2;
            min_dist2= (dist2<min_dist2)? dist2: min_dist2;

            j= 1;
            dist1=dist2=0;
          }

        dist3= dist4= 0;


      for(; j<K; j+=2)
          {

           #pragma acc loop vector
           for(c=0; c<D; c++)
               {
                  float punt1 = Punts[i*D+c];
                  float punt2 = Punts[(i+1)*D+c];
                  float centr1= Centroides[j*D+c];
                  float centr2= Centroides[(j+1)*D+c];

             //Let's calculate the distance from point i to centered j
              float temporal1 = punt1 - centr1;
              float temporal2 = punt1 - centr2;
              float temporal3 = punt2 - centr1;
              float temporal4 = punt2 - centr2;

              dist1 += temporal1*temporal1;
              dist2 += temporal2*temporal2;
              dist3 += temporal3*temporal3;
              dist4 += temporal4*temporal4;
          }

            min_arg1 = (dist1<min_dist1)? j     : min_arg1;
            min_dist1= (dist1<min_dist1)? dist1: min_dist1;

            min_arg2 = (dist3<min_dist2)? j     : min_arg2;
            min_dist2= (dist3<min_dist2)? dist3: min_dist2;

            min_arg1 = (dist2<min_dist1)? j+1    : min_arg1;
            min_dist1= (dist2<min_dist1)? dist1: min_dist1;

            min_arg2 = (dist4<min_dist2)? j+1    : min_arg2;
            min_dist2= (dist4<min_dist2)? dist4: min_dist2;
            }

            // centroid (min_arg1)  <--  point (i)
            Labels[i] = min_arg1;
            Sep[min_arg1]+=1;

        #pragma acc loop vector
            for (c=0; c<D; c++)
           //#pragma acc atomic
               New_Centr[min_arg1*D+c] += Punts[i*D+c];

            // centroid (min_arg2)  <--  point (i+1)
            Labels[i+1] = min_arg2;
            Sep[min_arg2]+=1;

        #pragma acc loop vector
            for (c=0; c<D; c++)
           //#pragma acc atomic
               New_Centr[min_arg2*D+c] += Punts[(i+1)*D+c];
          }
        // all points have been processed

      #pragma acc parallel loop gang vector_length(128)
      for (i=0; i<K; i++)
        {  // for every cluster
           int c;
         #pragma acc loop vector
         for (c=0; c<D; c++)
           {
           float value = New_Centr[i*D+c] / (float)Sep[i];

           // if centre is updated, set flag <r> to 1.
           if (value != Centroides[(i*D)+c])
               *pR = 1;

           // update with new centroid
             Centroides[i*D+c] = value;
           }
         }
     } // data present

   return r;
}

int main(int argc, char **argv){

      int N;                  //Number of points
      int D;                  //Dimensions of the point
      int K;                  //Number of centroides

      // string array to be read file name
      char* file = (char*)malloc(1024);
      // if all parameters don't given, set as default parameter
      if (argc < 5) {
            // Default parameters
            strcpy(file, "InputData_Large.csv");
            K = 50;
            N = 1000000;
            D = 68;

            printf("Usage: %s<InputData_Large.csv>, <K=Number of Clusters>,
<N=Number of Rows>, <D=Number of Columns>\n", argv[0]);
            printf("We now use default parameters with InputData_Large.csv,
K=%d, N=%d, D=%d\n",K,N,D);
      }
      else {
            // copy file name from argument to variable
            strcpy(file, argv[1]);
            K = atoi(argv[2]);
            N = atoi(argv[3]);
            D = atoi(argv[4]);
      }

      // allocate memory for points matrix, centres and so on
      float *Punts      = (float*) malloc( sizeof(float) * N * D );     //
Points matrix
      int   *Labels     = (int*)   malloc( sizeof(int)   * N );   // Centroid
assigned to Point
      float *Centroides = (float*) malloc( sizeof(float) * K * D );     //
Centroides
      float *New_Centr  = (float*) malloc( sizeof(float) * K * D );     //
New centroides
      int   *Sep        = (int*)   malloc( sizeof(int)   * K );       //
Number of points for each centroid
      int cont = 0;     // iteration count
      int final;  // flag of convergence
      int c;

      // allocate string array for caseid (row name)

      char** row_name = (char**)malloc(sizeof(char*) * N);
      for (c = 0; c < N; c++)
            row_name[c] = (char*)malloc(sizeof(char) * 10);

      printf("K-means:\n\t Points: %d\n\t Dimension: %d\n\t Centroides:
%d\n",N,D,K);

      // read input data from csv file
      if(read_CSV_data(file, row_name, Punts, N, D, K, Centroides) > 0)
            exit(1);

        #pragma acc data copyin(Punts[N*D]) copyout(Labels[N])
copy(Centroides[K*D])
        #pragma acc data create(New_Centr[K*D]) copyout(Sep[K])
      do {
        // reassignment each points to nearest centroids
        final = PointsToCentroides(Punts, N, D, K, Centroides, New_Centr,
Labels, Sep);
        cont += 1;
      }
```

```
    while (final && cont<1);

  printf("\nNumber of iterations needed (T): %d\n", cont);
  printf("\nNumber of changes (F): %d\n", final);

//exit(0);

  printf("\nNew centroides:\n\t");
  int i = 0, j = 0;

  for (i = 0; i<K; i++) {
        for (j = 0; j<D; j++)
              printf("%3.1f\t", Centroides[i*D + j]);
        printf("\n\t");
  }
  printf("\n");

  // get yellow line
  char line[1024];        // one line to be read from csv file
  FILE* fp = fopen(file, "r");  // read first line (caseid,....)
  fgets(line, sizeof(line), fp);
  fclose(fp);

  // print new clusters to result csv file
  FILE* result_file = fopen("result.csv", "w");
  if (!result_file) {
        printf("Can't create result file.\n");
        return 1;
  }
  fprintf(result_file, "Number of clusters: %d\n", K);
  for (c = 0; c < K; c++) {
  fprintf(result_file, "\tCluster-%d : %d elements\n", c + 1, Sep[c]);
  }

  for (i = 0; i < K; i++) {
        fprintf(result_file, "\nCluster - %d\n", i + 1);
        fprintf(result_file, "\t%s\n", line);

        for (j = 0; j < N; j++) {
              if (Labels[i] == i)
            {
              fprintf(result_file, "\t%s,", row_name[i]);
                for (c = 0; c < D; c++)
                  {
                fprintf(result_file, "%3.1f,", Punts[i*D+c]);
              }
              fprintf(result_file, "\n");
            }
              }
  }

  // close file
  fclose(result_file);
```

```
      // free allocated memory
      free(file);
      free(Punts);
      free(Centroides);
      free(New_Centr);
      free(Labels);
      free(Sep);
      free(row_name);

      exit(0);
      }
```

A.5 SLURM Script #1 for the Multi-Core CPUs and GPU

```
#!/bin/bash -l
#
#SBATCH --job-name=ML_KMs
#SBATCH -N 1 # number of nodes
#SBATCH -n 2 # number of cores TOTALES
#SBATCH --partition=cuda.q
#SBATCH --nodelist=aolin24
#SBATCH --gres=gpu:1

hostname
module add gcc/8.2.0
module load pgi/18.4
module add cuda/9.0

# list GPU (accelerator) info (PGI utility)
pgaccelinfo # For 0
export CUDA_VISIBLE_DEVICES=0,1
pgaccelinfo # For 0,1

# list GPU device info (CUDA utility)
nvidia-smi

#--- Number of CPU Cores

grep -c ^processor /proc/cpuinfo

#---Multi-core-CPU

pgcc -fast -acc -ta=multicore -Minfo=all kmeanACC.c -o km1
perf stat ./km1
pgprof ./km1 2>pgperf1.txt

#---GPU

pgcc -fast -acc -ta=tesla -Minfo=all kmeanACC.c -o km2
perf stat ./km2
pgprof ./km2 2>pgperf2.txt
nvprof --print-gpu-trace --print-api-trace ./km2 2>trace_all2.txt

exit 0
```

A.6 SLURM Script #2 for the NVIDIA Visual Profiler (nvvp) Tool

```
#!/bin/bash -l
#
#SBATCH --job-name=ML_KMs
#SBATCH -N 1 # number of nodes
#SBATCH -n 2 # number of cores TOTALES
#SBATCH --partition=cuda.q
#SBATCH --nodelist=aolin24
#SBATCH --gres=gpu:1

hostname
module add gcc/8.2.0
module load pgi/18.4
module add cuda/9.0

#---GPU

pgcc -fast -acc -ta=tesla -Minfo=all kmeanACC.c -o km3

nvprof --print-gpu-trace -o trace.txt ./km3

nvprof --analysis-metrics -o metrics.out ./km3

exit 0
```