INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH & DEVELOPMENT

ISSN 2278 – 0211 (Online)

Accelerating Data Mining Application in R Using CUDA C

Sneha Shankar
Technical Analyst, Credit Suisse, Mumbai, Maharashtra, India
Sharwari Gadkari
Associate Application Developer, Oracle Financial Services Software, Mumbai, Maharashtra, India
Himani Dudhat
Consultant, Fractal Analytics, Mumbai, Maharashtra, India
Nikita Chakraborty
M.B.A. Student, IIM, Lucknow, Uttar Pradesh, India
Radhika Somthankar
Student, Department of Electrical and Computer Engineering, Carnegie Mellon University, USA

Abstract:
This paper focuses on an innovative approach of implementing parallel processing using NVIDIA's Graphics Processing Unit (GPU) to accelerate a data mining application in R. In order to accomplish this, one of the most apposite and efficient solution is to use CUDA (Compute Unified Device Architecture). We have used the k-means clustering algorithm to demonstrate the effectiveness of CUDA C in terms of speed-up and reduced latency. It is a widely used unsupervised learning technique in data science applications. The currently existing sequential R programming technique using C for k-means algorithm was converted to a more optimized and efficient code that uses concepts of parallel computing using GPU. The efficiency of C and CUDA C codes has been compared on the basis of execution time.

Keywords: *CUDA, NVIDIA, GPU, Parallel processing, R, CUDA C, Clustering, k-means algorithm, Llyod algorithm, data mining*

1. Introduction

Due to the large amount of time taken for the implementation of the existing data analysis algorithms it has a detrimental impact on the overall efficiency of the data analysis process. One major reason for this is sequential execution of computationally heavy and memory intensive processes. R is open source software used in data mining which uses sequential execution methodology without conducive libraries for GPU computing. This results in increased latency and poor memory allocation in CPU which increases with the volume of the data and the complexity of the algorithm. The use of same instruction set repeatedly has increased the need of parallelism in this process. Thus, concurrent execution of these instructions using CUDA C can greatly improve the overall efficiency of the data analysis process.

In order to demonstrate the effectiveness of CUDA C for accelerating data mining applications, the process of clustering has been chosen that involves grouping of random data into clusters based on a relevant property. The main computational challenges in clustering are high level requirement for scalability, high dimensionality, incremental or stream clustering and insensitivity to input order.

GPU computing using CUDA C provides a promising solution to meet these requirements.

2. Theory

2.1. CPU versus GPU

As opposed to CPU which consists of few cores and processes code serially, a GPU has parallel architecture which comprises of thousands of smaller cores. These cores can thus handle multiple tasks simultaneously making the execution of code much more efficient. An illustration of this can be seen in Figure 1.

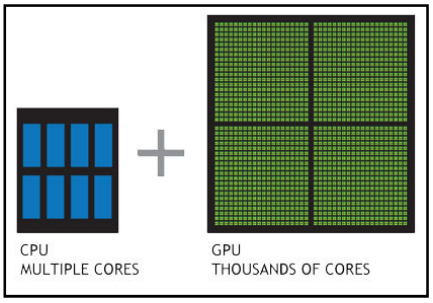

Figure 1: Cores Comparison (Reference-n VIDIA. (2014) CUDA C Programming Guide)

2.2. Accelerating Application Using GPU

The Graphics Processing Unit (GPU) is used along with the Central Processing Unit (CPU) in applications which require the acceleration of computationally expensive operations. Applications in the domain of engineering and analytics show a great potential for the use of GPU. They exhibit a drastically reduced execution time by decentralizing some of the compute-intensive portions to the GPU and keeping the remainder of the code still running sequentially on the CPU.

2.3 CUDA C

The Compute Unified Device Architecture (CUDA) is a computing platform which is compatible to the processing of General Purpose GPU (GPGPU). It is designed to be conducive with programming languages such as C, C++ and FORTRAN.

2.3.1. Programming Model

The highlights of the CUDA programming model and their method of implementation in C are presented as follows.

2.3.2. Kernels

The C programming language, when extended to CUDA C, enables the programmer to define and execute C functions, or more appropriately – kernels. The vast difference between functions in regular C and kernels are that the former allows only one execution when called, but the kernels are executed 'x' number of times in a parallel manner when 'x' different threads in the CUDA platform call them.

2.3.3. Thread Hierarchy

A thread is identified by its thread index. In CUDA, thread indexes are one or multi-dimensional. 1-D, 2-D, 3-D dimensional thread indexes are used to identify threads. Numerous threads together constitute to form a 'thread block'. This thread block can also be one-dimensional, two-dimensional or three-dimensional. These multi-dimensional threads and thread-block thus make computation possible across elements in a vector or matrix. In addition to a thread index, each thread has a thread ID too. They are closely related.

Block size	Thread Index	Thread ID
one-Dimensional	x	x
two-Dimensional (Dx, Dy)	(x, y)	(x + y Dx)
three-dimensional (Dx, Dy, Dz)	(x, y, z)	(x + y Dx + z DxDy)

Table 1: Relation between thread ID and thread index

All threads which form a block belong to the same processor core. This means that they have to share all the memory resources of the core. This imposes a constraint on the number of threads per block. Generally, in today's GPU, a thread block may be home to about 1024 threads. The architecture of the GPU is such that a kernel can be executed by more than one equally shaped thread blocks at the same time. Which means:

Total number of threads = number of threads per block * the number of blocks.

A 1-D, 2-D or 3-D grid of thread blocks are organized into blocks as illustrated by Figure 2.

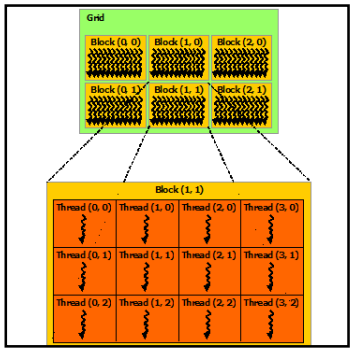

Figure 2: Grid of Thread Blocks (Reference-n VIDIA. (2014) CUDA C Programming Guide)

The number of processors which are part of the computation system and the corresponding data processed by them are used to calculate the number of thread blocks in a grid. Thread blocks are required to execute in parallel or series, but independently. This independence accounts for the flexible scheduling of thread blocks and can also take place in a multi-core processing environment. This gives scalability to the programmers as well.

2.3.4. Memory Hierarchy

Threads used for computation in the CUDA platform may access data from multiple memory spaces whilst they are in the midst of execution. Each thread has access to two types of memory: its private local memory and global memory as depicted in Figure 3. Every thread block has a memory which is shared by all threads of the block. This memory has the lifetime same as that of the block.

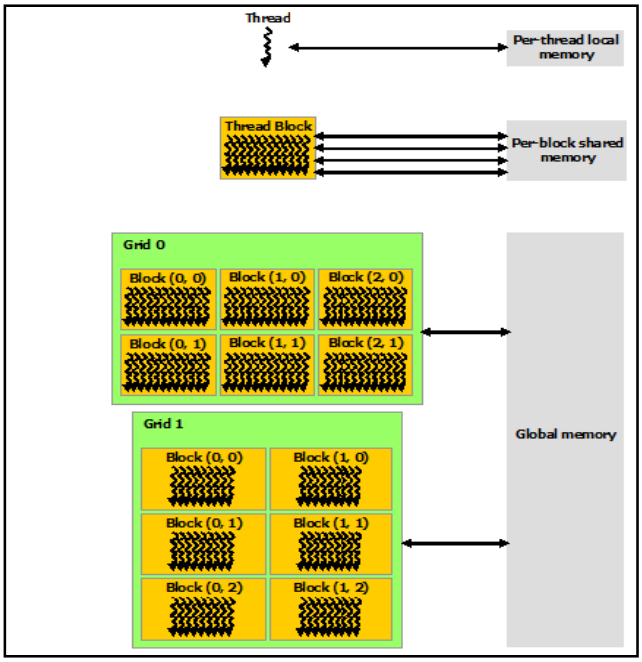

Figure 3: Threads multiple accesses (Reference-n VIDIA. (2014) CUDA C Programming Guide)

2.4. Clustering

Clustering is an unsupervised learning technique. It divides given data into a set of groups which are as similar as possible. A cluster is a collection of data objects which are similar or related to one another within the same group and dissimilar or unrelated to the objects in other groups. One commonly used basic algorithm for clustering is k-means algorithm.

2.4.1.*k*-Means Algorithm

k-means algorithm is also known as Lloyd's algorithm. For a set of 'n' observations, it divides the entire data space into 'k' clusters. Each of the 'n' observations belong to a particular cluster having the nearest mean. The nearest mean acts as a prototype of the cluster. Thus, the entire data space is clustered into what are called as Voronoi cells. A Voronoi diagram (Figure 4) partitions a plane into regions based on distance to predefined specific points called seeds in the plane. Each cluster has a centroid. Every centroid has a corresponding region in the data space which is closer to that centroid than any other centroid. This region is termed as a Voronoi cell. A Voronoi diagram is one that is created by taking pairs of closely situated data points and thereafter establishing a line that is equidistant between them as well as perpendicular to the line which connects them.

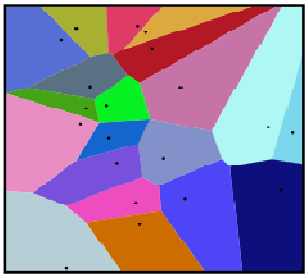

Figure 4: Voronoi Diagram (Reference-https://en.wikipedia.org/wiki/Voronoi_diagram)

Basic mechanism behind k-means algorithm can be simply explained as follows:
1. Decide the number of clusters k
2. Initialize the center of the clusters
3. Map each data point to its closest cluster
4. Calculate the position of each cluster to the mean of data points belonging to that cluster.
5. Repeat steps 3 and 4 until convergence.

This k-means algorithm eventually converges to a point. It is not required for this convergence point to be the least sum of squares. Here, the problem is non-convex and thus the algorithm may give convergence results which are restricted to local minimum. The algorithm ceases to compute which the clustered assignments do not change in consecutive iterations.

3. Implementation

The CUDA C program was developed in Visual Studio IDE and consequently files such as Exports Library File (.exp), Library File (.lib) and Dynamic Load Library (.dll) were generated using the NVIDA CUDA Compiler Command. The .dll file was then loaded in R from where the CUDA C code segment was invoked. The results were observed individually in Visual Studio as well as R.

4. Testing and Analysis

4.1. Observations

Sr. No.	C time	Cu time	Data Points	Clusters	VS/R	Speed up in %
1	120.1	18.48	10000000	12	VS	549.89
2	25.4	9.097	10000000	12	R	179.21
3	184.77	22.349	15000000	12	VS	726.74
4	26.237	7.726	15000000	12	R	239.59
5	242.78	22.869	20000000	12	VS	961.61
6	46.571	12.504	20000000	12	R	272.45
7	165.423	19.28	20000000	8	VS	758
8	28.183	11.374	20000000	8	R	147.78
9	513.276	28.112	20000000	15	VS	1725.82
10	260.167	28.182	15000000	15	VS	823.16
11	30.252	11.991	10000000	15	R	152.28
12	182.386	21.183	10000000	15	VS	761
13	103.137	12.267	10000000	8	VS	740.676
14	18.111	5.111	10000000	8	R	254.35

Table 2: Observations

Speed up = [(Time Taken by C) – (Time Taken by CU)/ (Time taken by CU)]

Several observations are recorded for different number of data points and clusters as seen in Table 2. It is observed that the efficiency in case of Visual Studio is more than that of R for the same set of input parameters (number of data points and clusters). This is because the execution time in the case of R is quite less as compared to Visual Studio, thus increasing the efficiency of the software. R's faster execution is primarily because of its systematic memory allocation and usage. R is designed as an 'in-memory' application: all of the data utilized for work must be hosted in the RAM of the machine on which the R application is running. This optimizes the performance and flexibility, but does place constraints on the size of data (since it must work all in RAM).

4.2. Graphical Representation

The following graphical representation gives a glimpse of the successful execution of both C and CUDA codes with 10 million data points and 12 clusters.

Figure 5 is a Vornoi diagram representing the discrete clusters (12 in this case) and the grouped data points. The data points are grouped into different clusters based on their minimum distances from the centroids of each cluster.

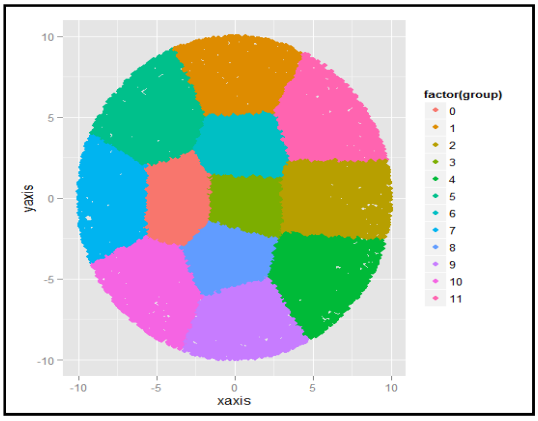

Figure 5: Scatter Diagram

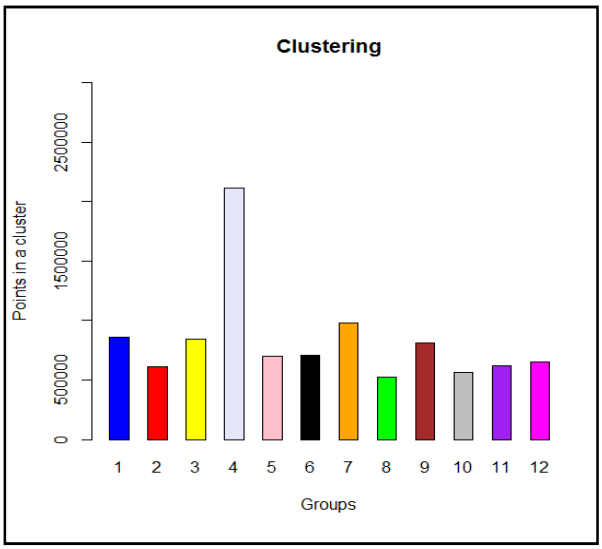

Figure 6: Cluster-wise grouping in C

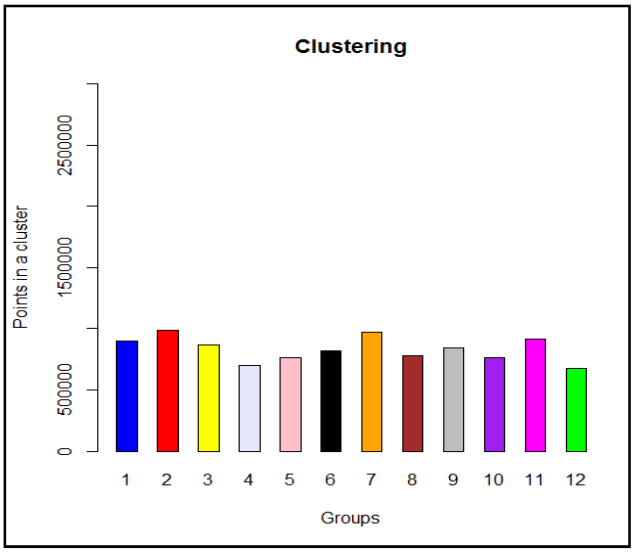

Figure 7: Cluster-wise grouping in CUDA C

The Figure 6 and Figure 7 are bar graphs showing the number of data points corresponding to each group in C and CUDA C respectively after clustering. The two graphs differ slightly because of the different functions used for random generation of the co-ordinates of each data point in C and in CUDA i.e. rand () in C and functions belonging to curand library in CUDA.

4.3. Results

4.3.1. Variation in Data Points

The speed-up of the application on increasing the number of data points on different platforms is as follows:

- Visual Studio:

Sr. No.	C time	Cu time	Data Points	Clusters	Speed up in %
1	120.1	18.48	1000000	12	549.89
2	184.77	22.349	1500000	12	726.74
3	242.78	22.869	2000000	12	961.61

Table 3: VS observations for change in data points

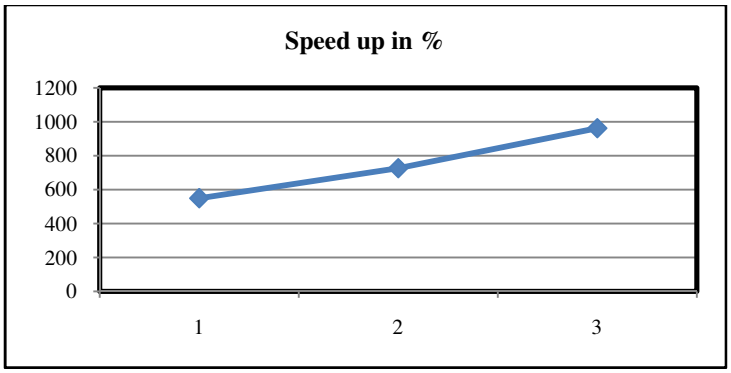

Figure 8: Speed up in VS

- R:

Sr. No.	C time	Cu time	Data Points	Clusters	Efficiency in %
1	25.4	9.097	1000000	12	179.21
2	26.237	7.726	1500000	12	239.59
3	46.571	12.504	2000000	12	272.45

Table 4: R observations for change in data points

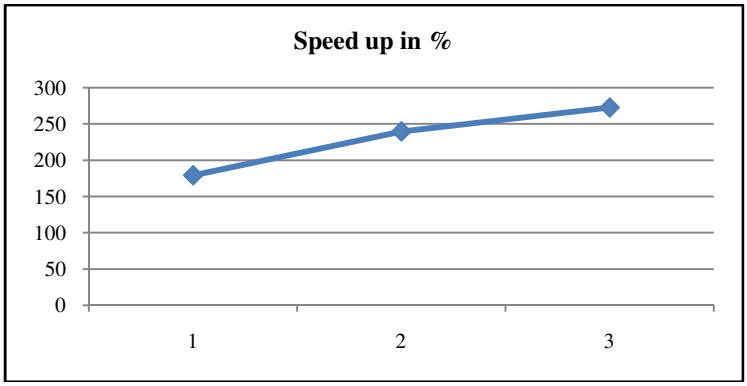

Figure 9: Speed up in R

4.3.2. Variation in Number of Clusters

The speed-up of the application on varying the number of clusters on both platforms is as follows:

- Visual Studio:

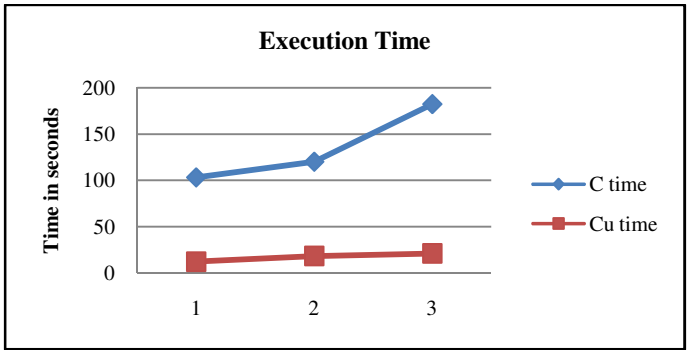

Figure 10: Efficiency of VS

Sr. No.	Centroids	C time	Cu time
1	8	103.137	12.267
2	12	120.1	18.48
3	15	182.386	21.183

Table 5: VS observation for change in clusters

- R:

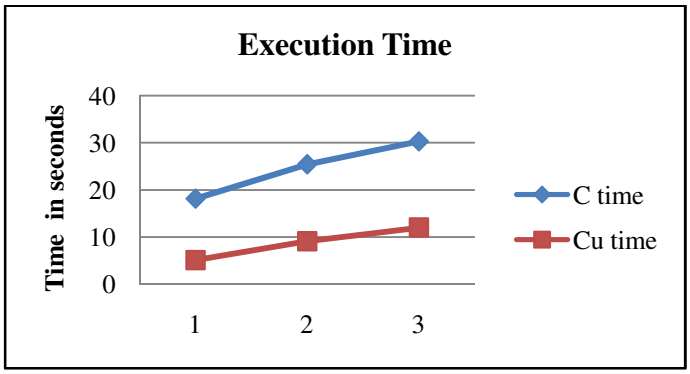

Figure 11: Efficiency of R

Sr. No.	K	C time	Cu time
1	8	18.111	5.111
2	12	25.4	9.097
3	15	30.252	11.991

Table 6: R observation for change in clusters

Thus, as the number of data points are increased:

- Rate of increase of time taken by C program more than CU program
- Efficiency of CUDA program better than C irrespective of platform
- Efficiency obtained is different for Visual Studio and R

As the number of clusters are increased:

- Computation increases considerably
- Rate of increase of execution time of CUDA very small as compared to C
- Efficiency of CUDA program better than C irrespective of platform

5. Conclusion

Thus, various sections of the code were successfully parallelized by inserting CUDA C kernels. We observed that the time taken for the code execution showed a drastic improvement over the sequential codes with high accuracy. The speed-up recorded was up to 700% using Visual Studio and 200% on the R platform. Thus, we can conclude that using GPU computing for data mining applications, the overall efficiency of the system can be improved manifolds and this grows with the volume of the data being processed. Thus, the applications working on large data sets can greatly benefit from this method.

As we have asserted before, the need for data mining in the industry and the volume of data is bound to increase in future. This calls for applications that can process data efficiently and accurately in reasonable time. Hence in the near future parallelizing data processing using GPU computation techniques might become a necessity.

6. References

i. Nicholas Wilt. (2011). The CUDA Handbook. Boston, MA: Addison-Wesley
ii. Jason Sanders, Edward Kandrot. (2011). CUDA by Example. Boston, MA: Addison-Wesley
iii. nVIDIA. (2014) CUDA C Programming Guide. Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4Kt1j8X4U
iv. nVDIA. (2011) CUDA C/C++ basics. Retrieved from https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
v. Winnie Thomas, Rohin D. Daruwala. (2014). Performance comparison of CPU and GPU on a discrete heterogeneous architecture. Circuits, Systems, Communication and Information Technology Applications (CSCITA), 2014. International Conference on (pp. 271-276). IEEE.DOI - 10.1109/CSCITA.2014.6839271
vi. ToonCalders. Data mining clustering. Retrieved from http://wwwis.win.tue.nl/~tcalders/teaching/datamining09/slides/DM09-08-Clustering.pdf
vii. nVIDIA. Retrieved from http://www.nvidia.com/object/what-is-gpu-computing.html#
viii. nVIDIA. Retrieved from http://www.nvidia.com/object/what-is-gpu-computing.html#
ix. C for Cuda - Small Introduction to GPU computing. (2013, May 14). Retrieved fromhttp://www.slideshare.net/IPALab/c-for-cuda-small-introduction-to-gpu-computing