Published in final edited form as:
J Supercomput. 2014 October ; 70(1): 284–300. doi:10.1007/s11227-013-0906-y.

High Performance Data Clustering: A Comparative Analysis of Performance for GPU, RASC, MPI, and OpenMP Implementations*

Luobin Yang[1], **Steve C. Chiu**[2], **Wei-Keng Liao**[3], and **Michael A. Thomas**[2]

Luobin Yang: yangluob@isu.edu; Steve C. Chiu: chiustev@isu.edu; Wei-Keng Liao: wkliao@ece.northwestern.edu; Michael A. Thomas: mthomas@isu.edu

[1]Department of Biological Sciences, Idaho State University

[2]Department of Electrical Engineering and Computer Science, Idaho State University

[3]Department of Electrical Engineering and Computer Science, Northwestern University

[2]Department of Biological Sciences, Idaho State University

Abstract

Compared to Beowulf clusters and shared-memory machines, GPU and FPGA are emerging alternative architectures that provide massive parallelism and great computational capabilities. These architectures can be utilized to run compute-intensive algorithms to analyze ever-enlarging datasets and provide scalability.

In this paper, we present four implementations of K-means data clustering algorithm for different high performance computing platforms. These four implementations include a CUDA implementation for GPUs, a Mitrion C implementation for FPGAs, an MPI implementation for Beowulf compute clusters, and an OpenMP implementation for shared-memory machines. The comparative analyses of the cost of each platform, difficulty level of programming for each platform, and the performance of each implementation are presented.

Keywords

Parallel Data Clustering; K-means Clustering; Scalability; Reconfigurable Computing; HPC

1. Introduction

The demand for performance – speed of computing – is never decreasing. The ever-increasing data size demands faster machines to process and analyze them [1]. Hardware accelerators such as Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) have emerged lately as promising technology drivers [2]. On the other hand, APIs such as MPI (Message Passing Interface) and OpenMP have traditionally provided a software-oriented approach.

*This research was partially sponsored by an Idaho National Laboratory Subcontract and an NIH INBRE Grant. See Acknowledgement.

This paper focuses on four implementations for the K-means data-clustering algorithm, using four different architectures, and provides a performance comparison for these differing implementations. Section 2 reviews the research and background work in GPU, FPGA and reconfigurable computing, MPI, OpenMP, and data clustering algorithms. Section 3 describes our research design for parallel algorithms for the four different architectures. Section 4 presents the design considerations for each of the parallel algorithms. Section 5 contains experimental results and discussion. Conclusions and future work are discussed in Section 6.

2. Background

2.1. Graphics Processing Unit (GPU) for General Purpose Computing with CUDA

The flexibility and performance demands have made GPUs change from the very specific graphics accelerator to more general purpose computing device. Compared to CPUs, the floating-point computational capabilities and the memory bandwidth of GPUs are approximately one order of magnitude greater [3]. Modern GPUs are cheap and ubiquitous. Many applications have shown orders of magnitude better performance than CPU computation. Recently, GPUs have been used to accelerate various non-graphics applications in many scientific areas that include bioinformatics, fluid mechanics, physics etc.

NVIDIA provides a programming tool called Compute Unified Device Architecture (CUDA) for programming GPUs. CUDA uses C-style syntax and it's an extension to C programming language. A typical program that uses GPUs has a host portion that runs on CPUs and device portion that are composed of kernels, which run on GPUs. The host portion controls the overall flow of the whole program and also controls the transfer of the data between GPUs and CPUs.

Our research attempted to gain insight on how data mining applications perform on GPU by implementing a classic data-clustering algorithm on it.

2.2. FPGA and Reconfigurable Computing

Chip-level integration of different types of electronics makes possible processors that combines computing, interconnect, and storage [4]. FPGA represents an emerging technology for computing platforms where vector processors and superscalar processors are connected through low-latency, high-bandwidth fabric, all within a single system [5]. The circuit elements of an FPGA are prefabricated, having been packaged and tested. Both the connections and the cells within an FPGA are programmable to achieve different logic functions. A typical programmable element of an FPGA chip consists of logic elements, programmable inter-connect points, and programmable switches, where a switch can realize various connections among the signals entering it. The cells are usually organized in a row-wise or grid-wise manner [6]. The general approach is to exploit FPGA's flexibility, e.g. configuring as many adders as a specific application may require per cycle, as versus the fixed number of adders provided in a typical CPU. Processing elements built from FPGAs, instead of ASICs (for Application-Specific Integrated Circuits), can be programmed for specific tasks. Orders-of-magnitude performance improvements have been observed on

some workloads over the use of conventional scalar microprocessors in large-scale parallel simulations [7]. The SGI white paper identified areas where FPGA-based reconfigurable computing system may be of value [8]: offloading of computation, data I/O with DMA, graphics and digital media (including DSP), and image recognition, among others. These are in addition to the traditional benefits already available from FPGA, i.e. new algorithms can be quickly prototyped and alterations made and evaluated in hardware. Our research attempted to build subroutines based on the emerging hybrid FPGA reconfigurable computing systems to enable the development of distributed clustering algorithms to help us better understand complex physical processes involving spatial structures across multiple scales. The work exploited the power of FPGAs in a reconfigurable supercomputer for enhanced performance and scalability.

2.3. Message-Passing Interface (MPI)

MPI [9] represents the specification of an API that allows computers to communicate with each other. It is widely used in computer clusters for parallel programming.

2.4. Open Multi-Processing (OpenMP)

OpenMP [10] is an Application Programming Interface (API) that was developed to support threaded-based multi-platform shared memory programming. The main features of this shared-memory programming model include: 1. All threads have access to the same, globally shared memory, 2. Data can be shared or private. 3. Shared data is accessible by all threads, 4. Data transfer is transparent to programmers, 5. Synchronization takes place implicitly [11].

2.5. Simple K-means Algorithm

In statistics and machine learning, K-means clustering is a method of cluster analysis which aims to partition *n* observations into K clusters in which each observation belongs to the cluster with the nearest mean. It is similar to the expectation-maximization algorithm for mixtures of Gaussians in that they both attempt to find the centers of natural clusters in the data. Given a set of observations ($\mathbf{x}_1$, $\mathbf{x}_2$, …, $\mathbf{x}_n$), where each observation is a *d*-dimensional real vector, K-means clustering aims to partition this set into K partitions (where K < *n*) as $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$ to minimize the within-cluster sum of squares [12].

3. Research Design

3.1. Parallel K-means Algorithm for GPUs

To achieve the best performance on GPU, the memory usage is a big part of consideration when designing an algorithm to run on GPU. Because the data transfer between the host memory and the device memory is expensive, we should try to avoid unnecessary memory transfer between the host memory and device memory as much as possible. There is a memory hierarchy in GPU and each memory level has a different speed. We should consider the memory access pattern of k-means algorithm and try to make the global memory access coalesced and the shared memory access with no or fewer bank conflict. For the data that are used often and stored on global memory, copying them to the shared memory can achieve much better performance.

There are two major computational parts in the simple K-means clustering algorithm, i.e. a distance calculation part to calculate the distance between each data point to each cluster center then searching the minimum distance between a data point to the cluster centers, and a cluster center updating part that calculates the new distance based on the data points that are assigned to a cluster center to find the new cluster center. For the distance calculation part, it fits the GPU architecture very well because there is no data dependence between calculating difference distances and searching the minimum distance for each data point so it can achieve the best speed-up. For the cluster center updating part, we need to sum up all the objects that belong to the same cluster center, if one thread is assigned to one data point, then all the threads that correspond to the data points that belong to the same cluster center, there will be a data dependence between these threads. If we assign one thread to sum up all the objects that belong to the same cluster center, then there will be no data dependence between threads. Since there are more data points than cluster centers, the cluster updating part achieves some speed-up but not as much as the distance calculation part.

There are other GPU implementations of K-means clustering, which include GPUMiner [13] and GUCAS-CU-Miner [14]. GPUMiner uses bitmap to represent the membership of data objects to clusters. Our method uses a parallel counting method to sum the number of objects in a cluster, which needs less storage space than bitmap storage but achieves similar performance. CU-Kmeans is the parallel implementation of k-means clustering algorithm on GPU in the GUCASCU-Miner system. It has three kernels that are Cluster label update, Centroid update, and Centroid movement detection. The Cluster label update and Centroid update kernels are quite similar to the corresponding kernels of our implementation. However, our implementation has a different centroid movement detection method. Their method uses the square error between old and new centroids but our method detects the membership change.

3.2. Parallel Algorithm with MPI

The MPI algorithm is based on the classical single program multiple data (SPMD) processing model, and it is assumed that the entire data sets are evenly partitioned among a given number, a.k.a. system size, of single-core processing nodes. These nodes form a subset or entirely a distributed memory cluster computer.

3.3. Parallel Algorithm with OpenMP

Our OpenMP algorithm, while based on the SPMD model also, is implemented on a shared-memory system in a thread-based manner. All the threads are aware of the entire data sets being processed. Due to the nature of the OpenMP API, locking of globals is implicitly handled by the optimizing compiler by way of the #pragma syntax.

3.4. Parallel Algorithm with FPGA

Our approach is to develop a parallel simple K-means clustering algorithm applicable to hybrid FPGA-based reconfigurable computing systems. The objectives are: (1) preliminarily characterize the RASC system; (2) develop a scalable and cost-effective algorithm; and (3) evaluate the developed algorithm for performance, extensibility and scalability. The evaluation involved two platforms, as described in the next two sections.

3.4.1. Evaluation Platform A: the MVP Simulator—The MVP simulator is a program provided by Mitrion C SDK. It simulates the real hardware executing of an application. Since it is time consuming to synthesize an application to the FPGA hardware, it is necessary to use the simulator when developing an application. The MVP simulator generates the exact results as FPGA hardware does when running an application. It is just simulated on a CPU so it takes much longer to run an application than running on a real FPGA hardware.

3.4.2. Evaluation Platform B: the RASC System—For this work we used the SGI Altix 4700, a RASC shared-memory computing cluster equipped with at least 2 FPGA units capable of uploading "processing kernels" specified in a Mitrion C source program [15]. The system used for our work contains 24 dual-CPU nodes of Intel Itanium processors, making the system effectively a 48-node 64-bit shared memory machine of reconfigurability to be provided by the FPGA units. However, it is noted that since the FPGA units currently emulate floating-point operations and data representations, developers generally dispatch integer "kernels" to the FPGA and resort to the CPU nodes for floating-point tasks. The CPU nodes are also responsible as host processors, which take care of the I/O and initialization work for most applications.

4. Parallel Simple K-means Algorithms

4.1. Parallel Implementation with CUDA

To maximum the parallel execution for K-means data clustering algorithm, we designed our algorithm so that for the distance calculation part, each thread is assigned to calculate one distance between one data point and one cluster center. For the minimum distance searching part, each thread is assigned to find the minimum distance to cluster centers for each data point. For the cluster center updating part, we assign each thread to calculate a new cluster center by summing up all the objects that belong to this cluster based on the minimum distance. A different design for the cluster center updating part is to assign one thread to each data point. Even though this design generates more parallel executing by having more threads, but this requires the summing up operation to be atomic so there is no race condition between threads. For compute compatibility less than 1.3, CUDA does not support atomic operations for double-precision floating- point numbers and other workarounds just make this cluster center updating perform poorly. Also having one thread to work on just one data point may not keep the thread busy enough if the dimension of the data points is not large enough, which means there are not enough computing operations for each thread to work on.

Compared to the high bandwidth of device memory located on GPU, the data transfer between the host memory and the device memory is expensive, we designed our algorithm to only transfer data twice between the host memory and the device memory. The first time is to transfer the data points from host memory to device memory and the second time is to transfer the final cluster centers and the membership from the device memory to the host memory.

There is a memory hierarchy in GPU and each memory level has a different speed. We designed our K-means data-clustering algorithm so that the global memory access is coalesced and there is no or fewer bank conflict when using the shared memory. Since the shared memory is much faster than the global memory, for the data that are used often and stored on global memory, copying them to the shared memory can achieve much better performance.

The CUDA code for the major loop of k-means data-clustering algorithm is shown in Figure 1. This major loop contains three kernel functions that are executed on GPU. These three kernel functions calculate the distance between data point and cluster centers, search the minimum distance to find the closest cluster center, sum up the membership change, and update the cluster center separately. The membership change summing kernel function is executed on GPU with a tree-like reduction technique. The CUDA code for the distance calculation kernel function is shown in Figure 2. In this kernel function, the shared memory was used to store the data points that are shared between the threads in a block.

4.2. Parallel Implementation with MPI

In the MPI algorithm, the whole data points are evenly partitioned and each partition is distributed to a processor. The cluster centers are broadcasted to each processor from processor 0. Each processor calculates the distance between the data points in its partition and the cluster centers. Using this distance, each processor determines which cluster center is closest to each data point and then assigns this data point to the cluster center. Then all the data points are summed for each cluster center by called the MPI all reduction function and then the new cluster centers are calculated. Figure 3 shows the code that's run on each MPI process and how the cluster centers are updated and synchronized between MPI processes.

4.3. Parallel Implementation with OpenMP

The parallel simple K-means algorithm using OpenMP looks very similar to the sequential version, especially when the atomic OpenMP pragma is used. Figure 4 shows the code for the distance calculation and cluster center update portion of the algorithm.

4.4. Parallel Implementation with FPGA

The Mitrion Virtual Processor has been developed for the purpose of allowing software developers to benefit from FPGA-based software acceleration, without having to deal with the complexities of hardware design [13]. This is achieved by way of a hardware abstraction layer (HAL) that serves as the interface between the high-level code and the hardware. The language environment, Mitrion C, provides traditional C-like compiler directives and tools for selecting either the simulator or the actual hardware for executing the target application. In this Section we detail the design of the parallel K-means algorithm to be adapted for RASC.

Such adaptation must consider the processing flow of the parallelized algorithm, the data dependencies if any between the interim results (from different processes), the actual hardware organization, and finally implementation with the Mitrion C code. Our approach included all of these design considerations. As shown in the following subsections, the

parallel K-means algorithm was coded in Mitrion C and executed in the MVP simulator first. Then a relatively minor header change was introduced into the developed code, along with a different make file, to create the executable for the RASC cluster.

4.4.1. Data Path Diagram—Figure 5 shows the data flow in a parallel computation. This data path diagram shows how the data are transferred in each iteration of the parallel K-means clustering algorithm. This data path diagram is a sub-graph of the overall data path diagram for the whole program. In this diagram, the purple nodes are inputs and outputs of this sub-graph. The grey nodes are the waiting nodes that are currently not performing any operations due to lack of data. Since this diagram was captured right before the program starts to execute, all the nodes are grey nodes besides the inputs and outputs. The data path diagram can show the dynamics of the data flow and when a parallel application is running, the nodes can change their colors to show what status they are in.

4.4.2. Design Considerations—There are two major steps in the K-means clustering algorithm: assign observations to the cluster and calculate the new cluster centers based on the assignment of the observations. When assigning observations to the cluster, there are two steps of calculations that should be done: calculate the Euclidian distance between each observation to each cluster and find the cluster center that is closest to the observation based on the distance calculated in the first step. The distance calculating part can be full parallelized since each distance is independent of any other distances. As long as there is enough processing element, each processing element can calculate one or more distances between one observation and one cluster center. Since there is no data dependence between each observation to find the closest cluster center for this observation, this step can be calculated in parallel for each observation.

To calculate the new cluster centers, all observations belong to the same cluster are summed and then divided by the total number of observations in the cluster along each dimension. For this step, there is no data dependence between each cluster, which means each new cluster center can be calculated in parallel. Usually the number of observations belong to a cluster can be big for some clusters, which means the summation calculation part can be accelerated provided a parallel summation can be implemented.

Since the cluster centers could change between consecutive iterations of the outermost loop of the algorithm, there is data dependence between iterations of the most outer loop. This means the outermost loop cannot be parallelized.

5. Results and Discussion

5.1. GPU results and discussion

We did two experiments using GPU implementation to see how our GPU implementation algorithm performs. First, we used different number of data points and measured the speedup of the GPU implementation compared to the sequential version. Second, we measured the execution time of the GPU implementation when the data size remains the same, but the number of clusters is different. The execution time measured is the wall-clock

time, which includes CUDA memory copy, CUDA memory allocation, and CPU and GPU time all together.

The relationship between the speedup and different number of data points is shown in Figure 6. Each data point has 10 dimensions and the number of clusters was fixed to be 256. When the number of data points is small, only a small portion of the GPU processors is utilized, and thus the speedup is small, as the number of data points increases, more and more GPU processors can be utilized and we thus observed more speedup. When the number of data points increased to a certain point, all GPU processors are utilized and the speedup is not increasing anymore. From Figure 4 we can conclude that our GPU implementation scales very well with the dataset size.

The relationship between the speedup and different number of clusters is shown in Figure 7. In this experiment, each data point has a dimension of 10 and the number of data points is fixed at 65536. The speedup increased as the number of clusters increases. This is because when the number of clusters is small, not all GPU processors can be utilized and when the number of clusters increases, more and more GPU processors can be utilized and thus the speedup increases. The speedup will stop to increase when all GPU processors are utilized. From Figure 5 we can conclude that our GPU implementation scales very well with the number of clusters when the number of data points is fixed.

For our GPU implementation, the running time was measured on a machine that has an AMD Phenom II Quad-core 3.0GHz processor and 8G RAM. The GPU used is NVIDIA Tesla C2050, which has 448 GPU cores and 3GB device memory. Tesla C2050 graphics card is attached to this machine, which runs Ubuntu 10.04 LTS server operating system.

5.2. MPI Results

To see the scalability of the MPI implementation, we ran the MPI program with a data set that contains 20000 data points (each data point is has 10 dimensions) and then measured the execution time of the MPI program with different number of processors. We set the number of clusters to be 100. The execution time used with different number of processors is plotted in Figure 8. We obtained this result using an Apple Xserve cluster with 24 compute nodes. Each compute node of the cluster had two 2.3GHz CPUs and 4GB RAM. The MPI implementation was compiled using the MPI library MPICH 1.2.6 library.

We performed a comparison between the sequential version and the MPI version. The speed-up with different computing nodes is shown in Figure 9. This figure shows a very good scalability of the MPI implementation when the number of processors increases.

5.3. OpenMP Results

We used the same data set as running MPI implementation to measure OpenMP implementation. The machine that we used to run the OpenMP implementation is a Linux box that has two quad-core Operon processors running at 2.3GHz. Figure 10 shows the running time of the OpenMP implementation when different number of threads is used. This figure shows a good scalability of the algorithm when the number of threads increases.

5.4. FPGA Results

5.4.1. Mitrion MVP Simulations—We have run the parallel K-means clustering program that we developed on the MVP simulator. We used a data set for image processing as the input. This data set contains 17692 observations and each observation has a dimension of 9. We have run the program with different number of clusters and then compared the results (cluster centers and membership of each observation) with those of the sequential program and we found there is no difference between the two execution results. This proved the correctness of our parallel K-means data-clustering algorithm.

Since the MVP simulator simulates the parallel program by running it sequentially, it is difficult to measure the actually scalability of our parallel K-means clustering algorithm on the MVP simulator, but it is very necessary to run a program on the MVP simulator before deploying and running it on the real FPGA hardware.

In our simulation runs, an input image data file, which contains 17692 data points and each data point is 9 dimensional, was used, and our parallel K-means algorithm was configured to generate 4, 8, 16, 32 and 64 clusters during separate executions. The simulator was run on an Apple Macbook laptop with a dual core Intel processor (2.0GHz). Figure 11 shows the executing time vs. number of clusters obtained from these runs. In this figure, the unit of the executing time is millisecond (ms).

5.4.2. SGI Altix 4700 Executions—The results from the RASC cluster conformed largely to those obtained from the MVP simulations. However, due to long time taken to register and upload the stream data and code before execution, the overall performance was somewhat reduced by this overhead.

Unlike the MVP simulation environment, on the RASC Altix 4700 cluster the compilation, linking, registering and data streaming needed to be performed in multiple phases. This was necessary to enable the allocation of hardware resources (i.e. configuration) including global and local RAMs. The complexity of hardware allocation is noted as a design tradeoff for the ease of programming in the high-level C-like language. As such, the “latency” in completing a computation run was increased.

5.5. Discussion on the Results

The results show very good scalability of each implementation. Even though each implementation shows scalability, the cost of the hardware is different. With the same number of processing units, the cost of GPU and FPGA is much less than that of a computer cluster or a shared-memory machine.

The performance of each implementation was measured on the assumption that the datasets fit in the memory of each platform. For instance, for the GPU implementation, all the data objects are loaded to the GPU device memory before the computation starts. If the size of the data objects exceeds the total memory of the device memory of GPU, the program will report an error of not enough memory. The performance of the GPU implementation increases with the size of data objects until enough threads are kept busy, after that point, the performance will decrease because of the overhead of thread dispatching.

From the perspective of the difficulty of programming, the development time for FPGA has been greatly reduced with the help of Mitron C, but the process of putting the image into the FPGA box still takes several hours for the platform we used. OpenMP is the easiest one because of the implicit parallelization of the compiler. MPI has a mature library that can be used to do reduction and many other basic parallel computing. Table 1 summarizes the pros and cons of each implementation based on the cost and the programming difficulty levels of the platforms.

6. Summary and Future Work

This paper presented our work in the parallelization of the classic simple K-means clustering algorithm for four different architectures. The MPI implementation running on a computer cluster and the OpenMP implementation running on a shared memory computer are traditional approaches. The GPU implementation and the FPGA implementation are newer approaches using new architectures. We measured the execution time for each parallel implementation and compared the performance of each implementation with the sequential version using the same data set. Each parallel implementation has demonstrated various levels of performance increase depending on the platform the implementation was designed for. We also measured the scalability of each implementation on a data set with different data points or with different number of clusters. Each implementation scales very well as the data points increases or as the number of clusters increases.

Our CUDA implementation, the computation task of all the points belonging to the same cluster is assigned to one CUDA thread, this scheme can be approved to achieve more balanced workload and scalability. With the Dynamic Parallelism support of the GK110 architecture [16], the computation task assigned to a thread can be parallelized and thus improve the scalability and balance the workload better.

For future work, we would like to explore additional data clustering algorithms that are frequently used as “processing primitives” for various scientific applications. Doing so provides us opportunities of gaining insight into the adaptability of the GPU and RASC FPGA systems for problems that are currently beyond reach with traditional parallel processing approaches and tools.

Given that the GPU is typically used for floating-point computation and FPGA for integer-based computation, it would be interesting to investigate the potential of their complimentary roles in tackling a complete application. Further more, as new languages and libraries become available, the authors would like to explore the benefits of paring FPGA, GPU and CPU in a holistic way. To the domain experts, ultimately such systems should become transparent, with a middleware-like layer addressing the access and scheduling for the entire computation load.

Acknowledgments

The authors would like to acknowledge the use of the SGI Altix 4700 located at Idaho National Laboratory for the work performed in this paper, and consultation with Dr. Charles Tolle for the data analysis of this project. The work is part of INL Subcontract/ISU No. 125-229-59. This work was also made possible by NIH Grant #P20 RR016454 from the INBRE Program of the National Center for Research Resources.

References

1. Hey, T.; Tansley, S.; Tolle, K. Microsoft Research. 2009. The Fourth Paradigm: Data-Intensive Scientific Discovery.
2. Sarkar, S.; Majumder, T.; Kalyanaraman, A.; Pande, P. Hardware Accelerators for Biocomputing: A Survey. IEEE International Symposium on Circuits and Systems (ISCS); 2012.
3. NVIDIA Corporation. NVIDIA CUDA programming guide, Version 2.3.1. 2009.
4. Schlesinger, TE. Information Storage and Nanotechnology; Keynote Speech at the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005); April 2005; Monterey, CA.
5. Dunning TH Jr. The Once and Future SciDAC. Journal of Physics: Conference Series. 2005; 16(2005)
6. Sarrafzadeh, M.; Wong, CK. An Introduction to VLSI Physical Design. McGraw-Hill; 1996.
7. Kindratenko, V.; Pointer, D. A Case Study in Porting a Production Scientific Supercomputing Application to a Reconfigurable Computer. Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006); 24–26 April 2006; Napa, CA.
8. Silicon Graphics, Inc. Extraordinary Acceleration of Workflows with Reconfigurable Application-specific Computing from SGI, the SGI White Paper. Nov. 2004
9. Message Passing Interface Forum MPI: A message passing interface standard. International Journal of Supercomputer Applications. 1994; 8(3/4):165–414.
10. OpenMP website. URL: http://openmp.org/mp/
11. Chapman, B.; Jost, G.; van der Pas, R. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press; 2007.
12. K-means clustering, definition of, Wikipedia page. URL: http://en.wikipedia.org/wiki/K-means_algorithm
13. Fang, W.; Lau, K.; Lu, M.; Xiao, X.; Lam, C.; Yang, P.; He, B.; Luo, Q.; Sander, P.; Yang, K. HKUST-CS08-07. 2008. Parallel Data Mining on Graphics Processors.
14. Jian L, Wang C, Liu Y, Liang S, Yi W, Shi Y. Parallel Data Mining Techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA). Journal of Supercomputing. 2011
15. Mitrionics Inc. The Mitrion C User's Guide. URL:http://forum.mitrionics.com/uploads/Mitrion_Users_Guide.pdf
16. NVIDIA Corporation. white paper of Kepler GK110 architecture. URL:http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

```
do {
        ++total_loops;

        CalculateDistance<<<blocksPerGrid,
threadsPerBlock>>>(d_objects, d_clusters,
d_membership, d_delta, nobjects, nclusters, nfeats);

        UpdateCluster<<<nblocks,
nthreads>>>(d_objects, d_clusters, d_membership,
nobjects, nclusters, nfeats);

        delta_total = thrust::reduce(delta_ptr, delta_ptr
+ nobjects);

        thrust::fill(delta_ptr, delta_ptr + nobjects, (int)
0);

        delta_ratio = 1.0 * delta_total / nobjects;

    } while (threshold < delta_ratio && total_loops <
  500);
```

Figure 1.
CUDA code for the major loop of K-means data-clustering algorithm

```
__global__ void CalculateDistance( const float * objects, const float * clusters, int * membership, int* delta, int nobjects, int nclusters, int nfeats )
{
        int idx = blockDim.x * blockIdx.x + threadIdx.x;
        int i, j;
        int tid = threadIdx.x;
        int sid = tid * nfeats;
        int cluster_id = -1;
        float diff, dist, min_dist = 1000000000000000000.0 ;

        __shared__ float sobj[2560];

        if ( idx < nobjects ) {
            // Load the object data point into shared memory.
            for (j = 0; j < nfeats; j++)
                sobj[sid+j] = objects[idx*nfeats+j];

            for (i = 0; i < nclusters; i++) {
                dist = 0.0;
                for (j = 0; j < nfeats; j++) {
                    diff = sobj[sid+j] - clusters[i * nfeats + j];
                    dist += diff * diff;
                }
                if ( dist < min_dist ) { min_dist = dist; cluster_id = i; }
            }
            if ( membership[idx] != cluster_id) {

                membership[idx] = cluster_id;
                delta[idx] = 1;
            }
        }
}
```

Figure 2.
CUDA code of the distance calculation kernel for K-means data-clustering algorithm

```
    MPI_Allreduce(&numObjs,      &total_numObjs,      1,
MPI_INT, MPI_SUM, comm);
do {
     delta = 0.0;
     for (i=0; i<numObjs; i++) {
          /* find the array index of nestest cluster center */
          index=find_nearest_cluster(numClusters,
numCoords, objects[i], clusters);

        /* if membership changes, increase delta by 1 */
              if (membership[i] != index) delta += 1.0;

              /* assign the membership to object i */
              membership[i] = index;

              /* update new cluster centers : sum of
objects located within */
             newClusterSize[index]++;
             for (j=0; j<numCoords; j++)
                  newClusters[index][j] += objects[i][j];
        }

        /* sum all data objects in newClusters */
```

```
        MPI_Allreduce(newClusters[0],          clusters[0],
numClusters*numCoords,MPI_FLOAT,          MPI_SUM,
comm);
        MPI_Allreduce(newClusterSize,          clusterSize,
numClusters, MPI_INT, MPI_SUM, comm);

        /* average the sum and replace old cluster centers
with newClusters */
        for (i=0; i<numClusters; i++) {
              for (j=0; j<numCoords; j++) {
                    if (clusterSize[i] > 1)
                          clusters[i][j] /= clusterSize[i];
                    newClusters[i][j] = 0.0;
               }
              newClusterSize[i] = 0;
        }

        MPI_Allreduce(&delta, &delta_tmp, 1, MPI_FLOAT,
MPI_SUM, comm);
         delta = delta_tmp / total_numObjs;
} while (delta > threshold && loop++ < 500);
```

Figure 3.
Code for MPI implementation

```
#pragma omp parallel for
                private(i,j,index)\
                firstprivate(numObjs)\
                private(numClusters,numCoords)\
                shared(objects,clusters,membership)\
                shared(newClusters,newClusterSize) \
               schedule(static) \
               reduction(+:delta)

for (i=0; i<numObjs; i++) {
   /* find the array index of nestest cluster center */
   index=find_nearest_cluster(numClusters,  numCoords,
objects[i], clusters);

   if (membership[i] != index) delta += 1.0;
   membership[i] = index;

    /* update new cluster centers : sum of objects
        located    within */

     #pragma omp atomic
     newClusterSize[index]++;

     for (j=0; j<numCoords; j++)
          #pragma omp atomic
            newClusters[index][j] += objects[i][j];
 }
```

Figure 4.
Code for OpenMP implementation

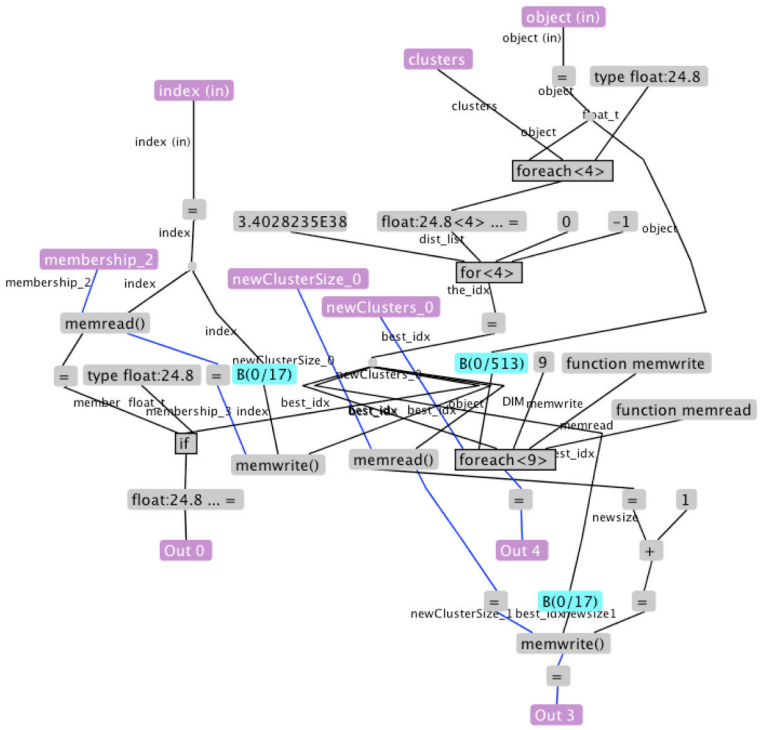

Figure 5.
Mitrion C data path diagram for parallel K-means clustering algorithm

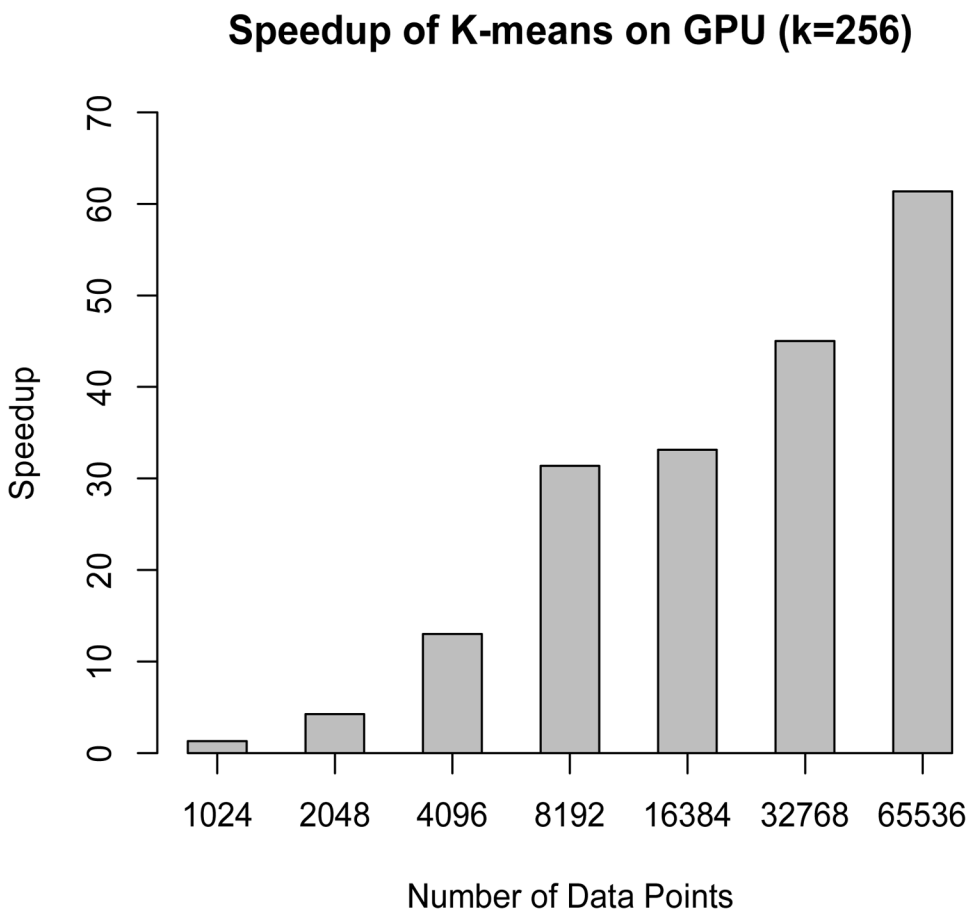

Figure 6.
Speedup vs. dataset size of GPU implementation

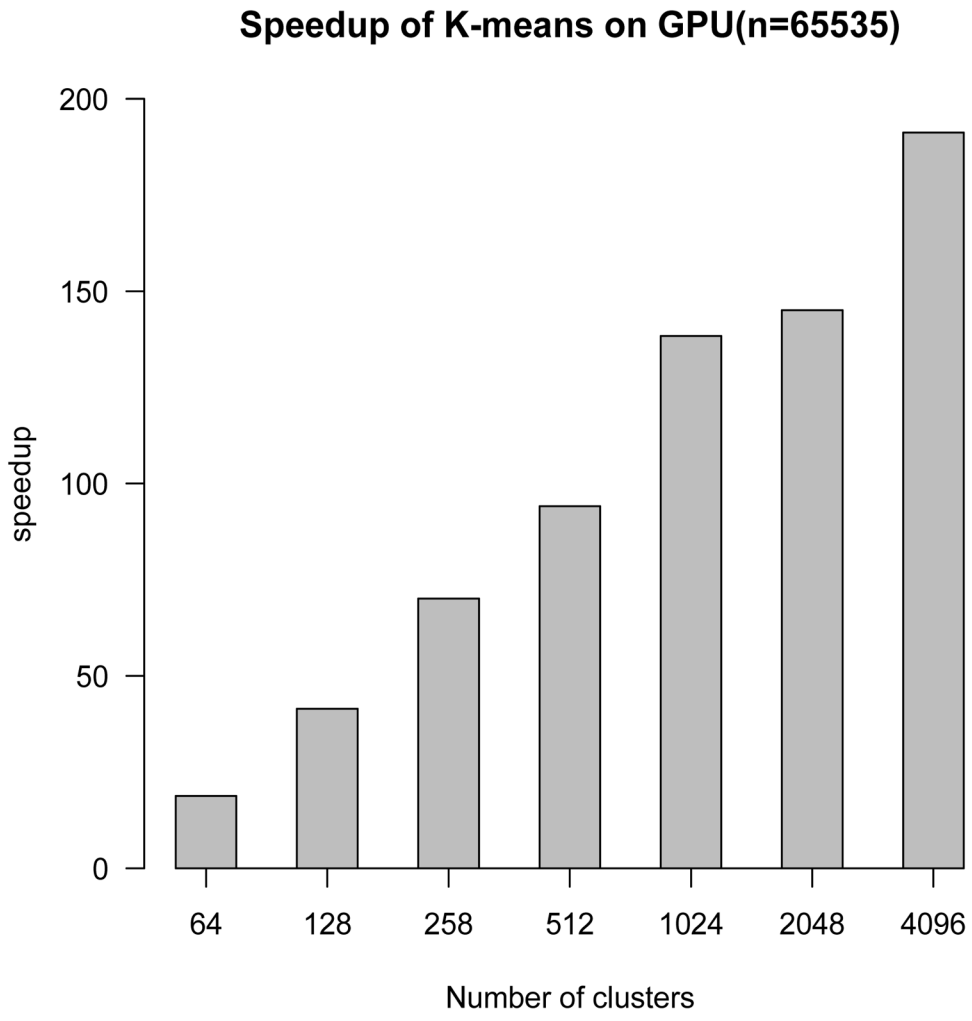

Figure 7.
Speedup vs. number of cluster centers of the GPU implementation

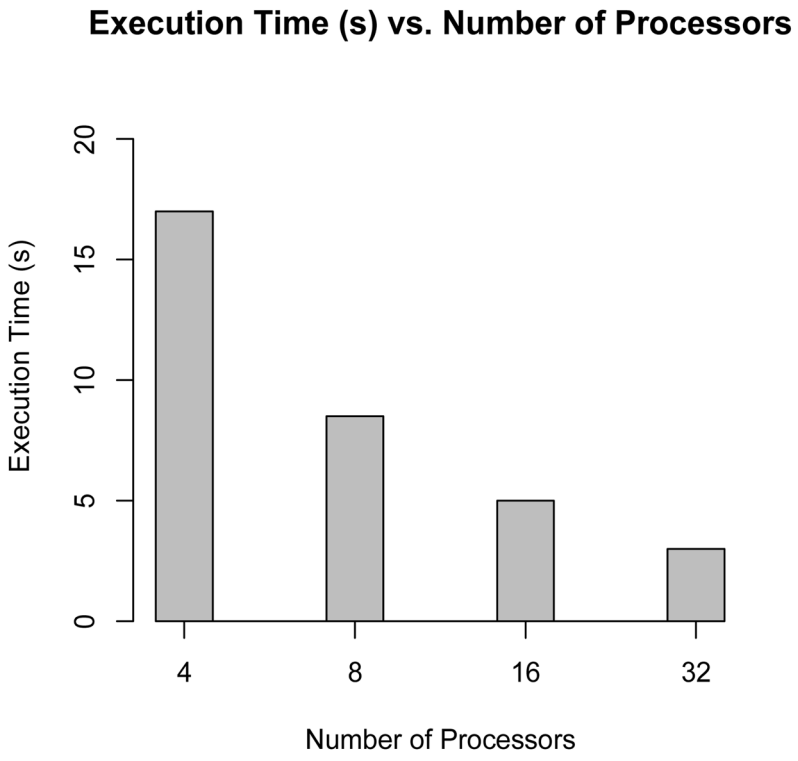

Figure 8.
Execution time vs. number of processors of the MPI implementation

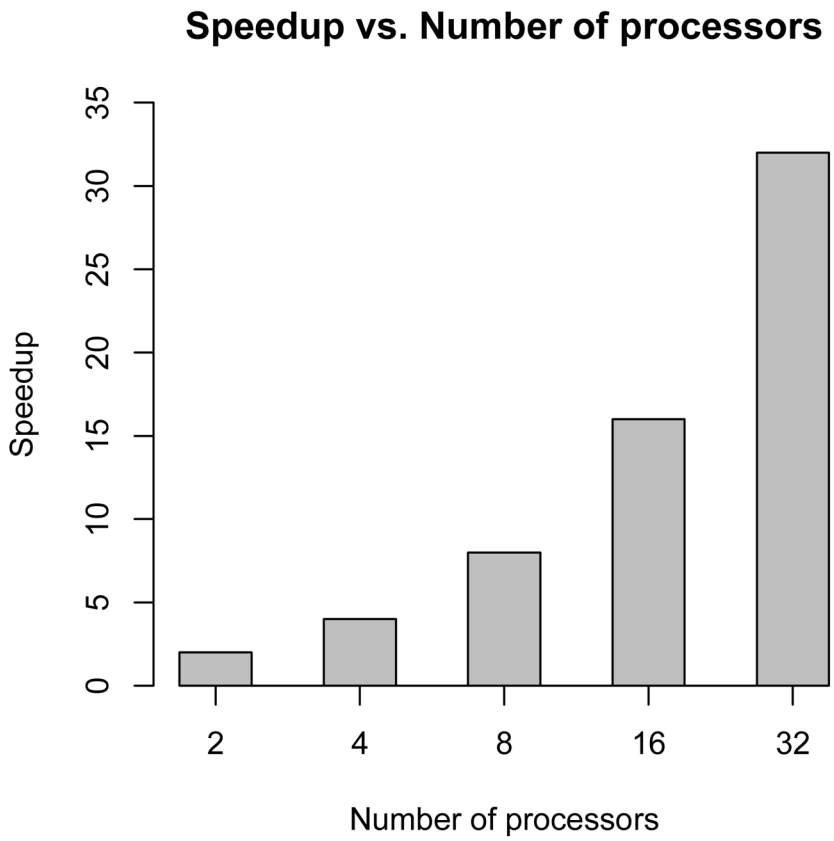

Figure 9.
Speedup vs. number of processors of the MPI implementation

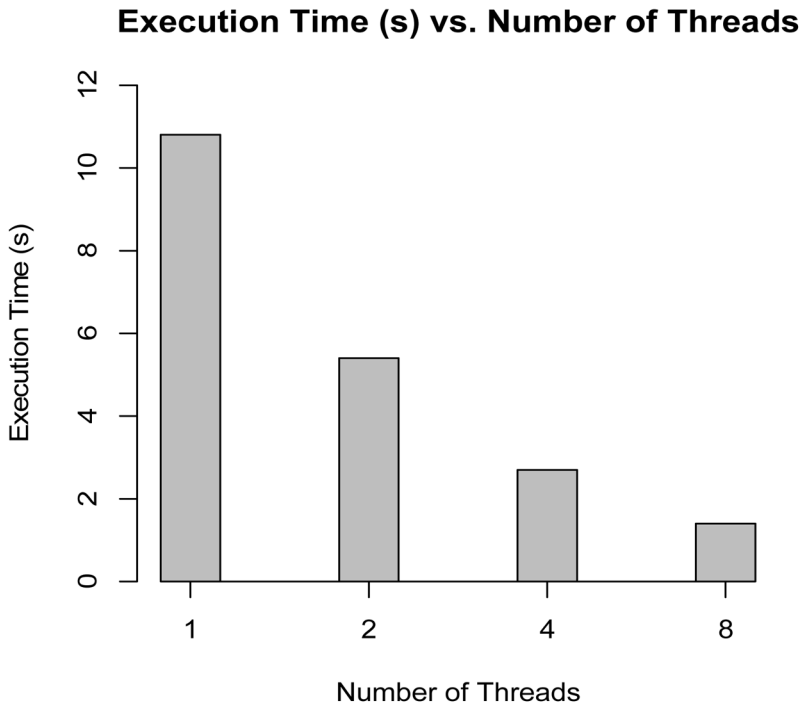

Figure 10.
Execution time vs. number of threads of the OpenMP implementation

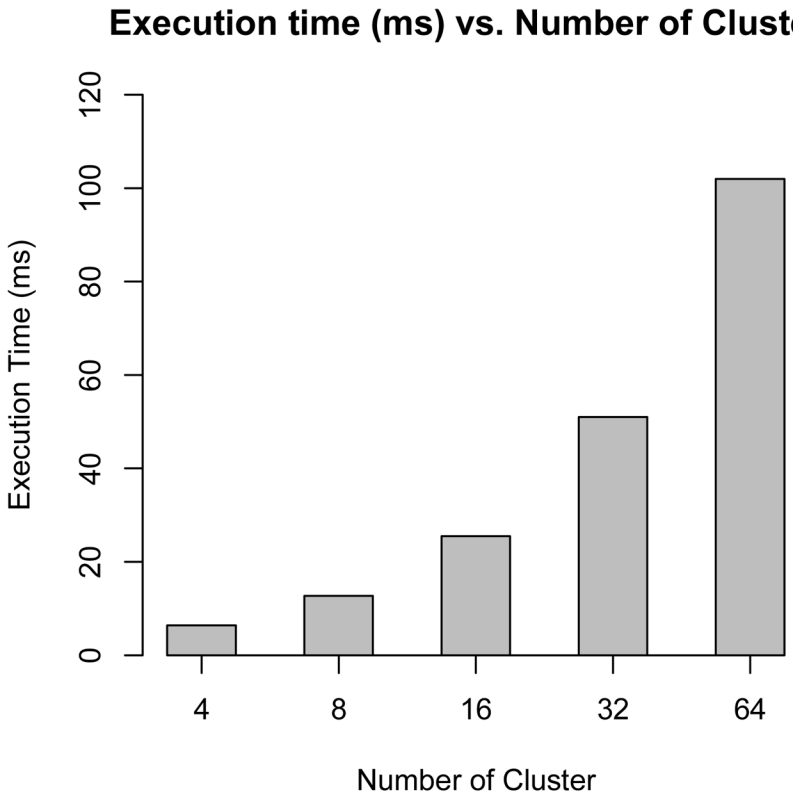

Figure 11.
Execution time vs. number of clusters of the FPGA implementation on MVP simulation

Table 1

Comparison of the four platforms for parallel programming

Platform	Pros	Cons
GPU	Low cost	Device memory is limited
FPGA	Low cost	Lack high-level programming language support
MPI	Mature library, programming is easier than GPU and FPGA	High cost
OpenMP	Medium cost, programming is easy	The total number cores is limited