Accelerating k-Means on GPU with CUDA Programming

Can Yang, Yin Li* and Fenhua Cheng
Hunan Vocational College of Science and Technology, Hunan, China

*Corresponding author

Abstract. We accelerate basic k-Means algorithm using CUDA GPU, a new programming model by NVIDIA, and experiment data shows we achieve a maximum speedup of 67.752, while other teams claim 20 to 40. Also we find that the basic k-Means algorithm is most sensitive to the cluster size k, and less to the datasets size b and least to the dimension d. In addition, we find the CUDA shared memory improves the performance, but also depends on which factor we scale.

Keywords: k-Means; GPU; CUDA.

1. Introduction
K-Means is a very popular clustering algorithm, which is widely used in fields like data analysis, pattern recognition and information retrieval. Basic k-Means algorithm is very simple. Following are the steps[1]-[3]:

- Randomly select initial centroids of k data points
- Repeat
- Assigning every data point to its closest centroid to form k clusters
- Recalculate each cluster’s centroid
- Until there are no changes of all the centroids

K-Means is a NP-hard problem[5]. If data points' number is n, the dimension of each point is d, and the number of clusters is k, then the time complexity is represented by \(O(n^d(dk+1) \log n)\).

This algorithm works well on small datasets, however, as the datasets grow bigger and bigger, it is becoming more and more necessary that the algorithm is scalable[5].

One of the solutions is parallel k-Means. Basically there are two approaches of parallelization, one is based on software platforms, such as MPI (Message Passing Interface) and OpenMPI, the other one is based on hardware accelerators, like FPGA (Field Programmable Gate Array) and GPU (Graphics Processing Unit)[6].

There is a lot of research on how to speed up k-Means with GPU[7]-[9]. GPU_Miner makes full use of global memory and shared memory, adopts the design of a bitmap and has a block size of 128(pieces), while bitmap consumes too much memory[10]. UV_k-Means use texture memory instead for data storage and has a block size of 156[11]. Dimension-Adaptive k-Means uses register most for low-dimensional datasets and heavily relies on shared memory for high-dimensional datasets, but never uses texture memory[12].

2. Our Solutions
In line 3 of the basic k-Means above, each data point can find its closest centroid concurrently, there is no data dependency. So we can parallelize it using CUDA threads. Most of our work is to implement
the CUDA version of k-Means and figure out how sensitive that k-Means is to the 3 factors: the datasets size \( n \), the data dimension \( d \) and the number of clusters \( k \).

2.1. Handle Large Datasets
Previous research assumes that all the datasets could be filled in the device memory, while we support more large datasets by doing GPU computation in batch.

2.2. Compare Shared Memory and Global Memory Performance
We try to speed up the CUDA k-Means further by memory optimization. We provide two CUDA kernel functions, one uses the global memory to reference the cluster centroid, the other uses the shared memory. The only differences are at the beginning and how we access the data later.

Below are the algorithms for our program:

```c
/* Host drive function */
Function HostDrive()
Begin
  Randomly select the initial cluster centroids of \( k \) data points
Repeat
  Split the data points into \( S \) subsets
  For each subset of data points in \( S \)
    From the host memory copy the subset of data points to the device global memory
    From the host memory copy the \( k \) initial cluster centroids to the device global memory
    Start kernel function of CUDA in GPU device
    From the device global memory copy the cluster indices each point belongs to the host memory
  End for
  Update the new centroid of each cluster
Until centroids do not change
End function
```

```c
/* Device kernel function, not using the shared memory */
Function Device Kernel-No-Share()
Begin
  \( M = \text{max point index in a batch} \)
  // blockIdx, blockDim and threadIdx is CUDA predefined variables, offset is the data point index that a GPU thread corresponds to
  offset = blockIdx.x * blockDim.x + threadIdx.x
  If offset < \( M \)
    Get cluster centroids from the global memory
    Calculate the distance to each cluster centroid
    Find the smallest distance
    Update the cluster index the data point is associated to
End function
```

```c
/* Device kernel function, using the shared memory */
Function Device Kernel-Share()
Begin
  \( m = \text{max point index in a batch} \)
  // \( k \) is the number of clusters, total = \( k \times d \)
  for (pos = offset; pos < total; pos += blockDim.x)
    Copy centroids[pos] from global memory to shared memory
End for
//call CUDA thread synchronization function
```
//make sure the cluster centroids are loaded
__syncthreads()
// blockIdx, blockDim and threadIdx is CUDA predefined variables
//offset is the data point index that a GPU thread corresponds to
offset = blockIdx.x * blockDim.x + threadIdx.x
If offset < M
Get cluster centroids from the shared memory
Calculate the distance to each cluster centroid
Find the smallest distance
Update the cluster index the data point is associated to

End function

3. Results

3.1. Experiment Environment

We do experiments on Amazon EC2, and the instance type is g2.2xlarge. Below are the details about hardware information including CPU and GPU.

- **CPU** There are 8 vcpu (defined by EC2), the specifications of each is as follows:
  - Model name: Intel(R) Xeon(R) CPU E5-2670
  - CPU MHz: 2593.846
  - Cache size: 20480 KB

- **GPU** The GPU device is NVIDIA GRID K520, and the specifications related to the CUDA programming mode are as follows:
  - (8) Multiprocessors, (192) CUDA Cores/MP: 1536 CUDA Cores
  - GPU Max Clock rate: 797 MHz(0.80 GHz)
  - Total amount of global memory: 4036 MBytes
  - Memory Bus Width: 256-bit
  - Memory Clock rate: 2500 MHz
  - L2 Cache Size: 524288 bytes
  - Total amount of shared memory per block: 49152 bytes
  - Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
  - Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

3.2. Experiment Data

We do 2 types of experiments: the first type is comparing the performance between the serial version and the CUDA version. The second is to test the memory optimization in CUDA. Below are the experiment data collected. Elapsed time is measured in seconds.

(1) Serial versus CUDA

| d=2,k=10 | n     | 100,000 | 200,000 | 400,000 | 800,000 | 1,600,000 | 3,200,000 |
|----------|-------|---------|---------|---------|---------|-----------|-----------|
| Serial   | 1.997 | 4.145   | 6.477   | 20.068  | 48.237  | 87.472    |
| CUDA     | 0.445 | 0.628   | 0.820   | 1.820   | 3.966   | 8.258     |
| Speedup  | 4.486 | 6.604   | 7.902   | 11.028  | 12.162  | 10.59     |

**Table 1.** Varying data size.

| n=20000,k=10 | d=2 | 4   | 8   | 16  | 32   |
|---------------|-----|-----|-----|-----|------|
| Serial        | 0.458 | 1.720 | 4.855 | 7.875 | 20.009 |
| CUDA          | 0.341 | 0.366 | 0.930 | 2.760 | 5.341 |
| Speedup       | 1.343 | 4.699 | 5.220 | 2.853 | 3.746 |

**Table 2.** Varying data dimension.
Table 3. Varying cluster size.

| n=60000, d=2 \ k | 10   | 20   | 40   | 80   | 160  | 320  |
|------------------|------|------|------|------|------|------|
| Serial           | 1.001| 5.774| 18.061| 10.805| 43.123| 65.754|
| CUDA             | 0.401| 0.518| 0.830| 0.626| 0.797| 0.971|
| Speedup          | 2.500| 11.157| 21.759| 17.251| 54.100| 67.752|

CUDA Memory optimization

Table 4. Varying data size.

| d=2, k=10 \ n  | 10,000 | 20,000 | 40,000 | 80,000 | 1,600,000 | 3,200,000 |
|----------------|--------|--------|--------|--------|-----------|-----------|
| No-share       | 0.445  | 0.628  | 0.820  | 1.820  | 3.966     | 8.258     |
| Share          | 0.375  | 0.384  | 0.584  | 0.677  | 2.576     | 5.424     |
| Speedup        | 1.187  | 1.635  | 1.404  | 2.688  | 1.540     | 1.522     |

Table 5. Varying data dimension.

| n=20000, k=10 \ d  | 2     | 4     | 8     | 16    | 32      |
|--------------------|-------|-------|-------|-------|---------|
| No-share           | 0.341 | 0.366 | 0.930 | 2.760 | 5.341   |
| Share              | 0.313 | 0.334 | 0.900 | 2.686 | 5.283   |
| Speedup            | 1.089 | 1.096 | 1.033 | 1.028 | 1.011   |

Table 6. Varying cluster size.

| n=60000, d=2 \ k | 10   | 20   | 40   | 80   | 160  | 320  |
|------------------|------|------|------|------|------|------|
| No-share         | 0.409| 0.613| 0.550| 0.760| 1.080| 1.352|
| Share            | 0.406| 0.417| 0.416| 0.432| 0.473| 0.476|
| Speedup          | 1.007| 1.470| 1.322| 1.759| 2.283| 2.840|

4. Analysis and Conclusion

4.1. Serial and CUDA

As we can see from Table 1 to Table 3, CUDA performs much better than the serial k-Means. It achieves a maximum speedup of 67.572(times), which is better than what UV k-Means achieved, whose speedup of twenty to forty. But actually the speedup depends on what we scale. By increasing the datasets size, we only get a maximum of 12.162. And we get even less if we only increase the dimension. From Figure 1, Figure 3 and Figure 5 we can see that the serial k-Means is most sensitive to d, and less to k and least to n. Figure 2, Figure 4 and Figure 6 demonstrate how many times of computing speed CUDA versus Serial CPU in varying the 3 parameters n, d and k, separately.

Figure 1. Serial-CUDA: Varying the datasets size n – time.

Figure 2. Serial-CUDA: Varying the datasets size n – speedup.
Figure 3. Serial-CUDA: Varying the data dimension \(d\) – time.

Figure 4. Serial-CUDA: Varying the data dimension \(d\) – speedup.

Figure 5. Serial-CUDA: Varying the cluster size \(k\) – time.

Figure 6. Serial-CUDA: Varying the cluster size \(k\) – speedup.

On the contrary, the CUDA k-Means is most sensitive to the dimension \(d\), less to the datasets size \(n\) and least to the cluster size \(k\). The plots above also show this. In Figure 5, the CUDA curve is almost horizontal; in Figure 3, the CUDA curve goes up much more quickly when doubling \(d\).

4.2. CUDA Memory Optimization

Also from Table 4 to Table 6, we can see shared memory does help, and again we get the maximum speedup 2.84(times) from memory optimization. Figures below are the experiment data comparing Share Memory and No-share Memory computing by CUDA.

Figure 7. CUDA memory optimization: Varying the datasets size \(n\) – time.

Figure 8. CUDA memory optimization: Varying the datasets size \(n\) – speedup.
5. Conclusion
From the experiment results mentioned above, we can find out that GPU is a very effective accelerator hardware for k-Means algorithm, and the speedup depends on how we scale the datasets and the number of cluster k. GPU shared memory also helps but the extent also depends on these 3 factors: n, k and d. For future work, in data-intensive scenarios, data movement between CPU and GPU also takes nonnegligible time. How to change the algorithms to accommodate data, so that the algorithms are scalable in big data is very important.

Acknowledgments
This work received funding from the “Research on Technology Selection of Abnormal Surface Monitoring of Canal Slope between GNSS and Machine Vision”, Hunan Provincial Education Department Scientific Research Project, under Grant 19B227.
References

[1] Apostolakis J. An Introduction to Data Mining. Data Mining in Crystallography. Springer Berlin Heidelberg, 2009.

[2] Kanungo, T, Mount, D.M, Netanyahu, N.S, et al. An Efficient k-Means Clustering Algorithm: Analysis and Implementation. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2002, 24(7):881-892.

[3] Marco Capóa, Aritz Péreza, Lozanoa J A . An Efficient Approximation to the k-Means Clustering for Massive Data. Knowledge-Based Systems, 2016, 117:56-69.

[4] Mahajan M, Nimhorkar P, Varadarajan K, et al. The planar k-means problem is NP-hard. Theoretical Computer Science, 2012: 13-21.

[5] Bachem O, Lucic M, Krause A, et al. Scalable k-Means Clustering via Lightweight Coresets. arXiv: Machine Learning, 2017.

[6] Yang, Luobin, Chiu, et al. High performance data clustering: a comparative analysis of performance for GPU, RASC, MPI, and OpenMP implementations. The Journal of Supercomputing, 2014, 70(1):284-300.

[7] David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach, 2016, 3rd edition.

[8] Bhimani J, Leeser M, Mi N, et al. Accelerating K-Means clustering with parallel implementations and GPU computing. IEEE High Performance Extreme Computing Conference, 2015: 1-6.

[9] Xiao Y, Feng R, Lueung C, et al. GPU Accelerated Spherical K-Means Training. International Conference on Neural Information Processing, 2013: 392-399.

[10] W. Fang, K. K. Lau, M. Lu, et al. Parallel Data Mining on Graphics Processors. Technical Report HKUSTCS08, 2008.

[11] S. Che, M. Boyer, J. Meng, et al. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. Journal of Parallel and Distributed Computing, 2008, 68(10): 1370-1380.

[12] Li Y, Zhao K, Chu X, et al. Speeding up k-Means algorithm by GPUs. 10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010. IEEE, 2010.