A Parallel Implementation of the Gustafson-Kessel Clustering Algorithm with CUDA

Jeong Bong SEO†, Nonmember and Dae-Won KIM†a, Member

SUMMARY Despite the benefits of the Gustafson-Kessel (GK) clustering algorithm, it becomes computationally inefficient when applied to high-dimensional data. In this letter, a parallel implementation of the GK algorithm on the GPU with CUDA is proposed. Using an optimized matrix multiplication algorithm with fast access to shared memory, the CUDA version achieved a maximum 240-fold speedup over the single-CPU version.

key words: clustering, Gustafson-Kessel, CUDA, GPU

1. Introduction

A programmable graphics processing unit (GPU) provides a large number of simple multi-threaded cores supporting massively data-parallel computations. To achieve general-purpose parallel computation on GPUs, the Compute Unified Device Architecture (CUDA) has been developed to provide a set of APIs, compilers, and libraries for thread creation, synchronization, and data allocation on GPUs. Moreover, CUDA does not require reformulating algorithms to incorporate graphics concepts such as the graphics pipeline and 3D APIs.

The rapid advance in GPUs using CUDA has made it possible to parallelize clustering algorithms on desktop computers, offering dramatic speedups compared to conventional counterparts. Che et al. [1] presented the first CUDA version of the k-means algorithm in which the clusters of data points are partitioned into thread blocks, with each thread associated with one data point. The task of searching for the nearest centroid to each data point is completely independent, as is the task of computing a new centroid for each cluster. The CUDA version achieved a 72-fold speedup compared to the single-CPU version. Afterward, several variants of the parallel k-means algorithm using CUDA have been proposed and provided faster performance than the traditional CPU implementations [2]–[5].

In the present study, a parallel implementation of the Gustafson-Kessel (GK) clustering algorithm on a GPU using CUDA is proposed. Conventional k-means-type algorithms are limited to identifying different shapes of clusters because they use a fixed distance norm when calculating the distance between data points. In contrast, the GK algorithm uses an adaptive distance norm, described in terms of the covariance matrix for each cluster in which the eigenstructure is exploited, to identify the shape of the cluster [6]; it has shown better performance in various areas [7]–[9]. Despite the benefits of the GK algorithm, it becomes computationally inefficient when applied to high-dimensional data; in each iteration step, a covariance matrix for each cluster is computed and subsequently inverted, and in addition, its determinant is computed. To reduce the computational costs of the GK algorithm, a CUDA-based parallel implementation of the GK algorithm is presented here, using an optimized matrix multiplication approach with fast access to shared memory.

2. A Parallel Gustafson-Kessel Algorithm with CUDA

2.1 Gustafson-Kessel Algorithm

The GK algorithm generates a partition that provides a degree of membership for each data point in a given cluster. Given a data set \( Z = \{Z_1, Z_2, \ldots, Z_N\} \) in \( f \)-dimensional space, a number of clusters \( c \), and the cluster volumes \( \rho_i = 1 \), the objective function of the GK algorithm finds \( c \) disjoint and homogeneous clusters \( P = \{P_1, P_2, \ldots, P_c\} \) where \( V = \{V_1, V_2, \ldots, V_c\} \) is a vector of cluster centroids:

- **S0. Initialize membership** \( \mu_{ik}^{(t)} \) \( (t \leftarrow 1) \) of \( Z_k \) belonging to \( P_i \) for \( 1 \leq i \leq c, 1 \leq k \leq N \) such that \( \sum_{t=1}^{\infty} \mu_{ik} = 1.0 \).
- **S1. Update the centroids** \( V_i = \{V_{i1}^{(t)}, V_{i2}^{(t)}, \ldots, V_{ic}^{(t)}\} \) for \( 1 \leq i \leq c \):
  \[
  V_{it}^{(t)} = \frac{\sum_{k=1}^{\infty} \mu_{ik}^{(t-1)} Z_k}{\sum_{k=1}^{\infty} \mu_{ik}^{(t-1)}}
  \]
  (1)
- **S2. Update cluster covariance matrices** \( M_i \) for \( 1 \leq i \leq c \):
  \[
  M_i = \frac{\sum_{k=1}^{\infty} \mu_{ik}^{(t-1)} (Z_k - V_i^{(t)}) (Z_k - V_i^{(t)})^T}{\sum_{k=1}^{\infty} \mu_{ik}^{(t-1)}}
  \]
  (2)
- **S3. Compute the distances between** \( Z_k \) and \( V_i^{(t)} \):
  \[
  D_{ik}^2 = (Z_k - V_i^{(t)})^T [\rho_i \det(M_i)]^{1/2} M_i^{-1}(Z_k - V_i)
  \]
  (3)
- **S4. Update the membership of** \( Z_k \) at \( t \):
  \[
  \mu_{ik}^{(t)} = \left( \sum_{j=1}^{c} \frac{D_{ik}^2}{D_{jk}^2} \right)^{-1}
  \]
  (4)

By repeating Steps 1 to 4, the GK algorithm iteratively improves cluster partitions until no further improvement is
possible. Of the five steps of the GK algorithm, Step 2 is the most time-dominant task, which takes $O(cNf^2)$; this time complexity is computationally prohibitive when applied to high-dimensional data ($f$ is very large). Thus, the present study focuses on an optimized parallel implementation of Step 2 using CUDA. Similarly to the work of Espenshade et al. [5], the parallelization of the other steps was implemented in a straightforward manner.

2.2 A Parallel Gustafson-Kessel Algorithm

To implement the GK algorithm with CUDA, issues of data-localization, memory access, and efficient parallelism needed to be addressed. It is important to speed up inter-thread communication within thread blocks by localizing data access patterns. To this end, in the present study, the PBSM (per-block shared memory) was chosen as the storage area for the data points and centroids. Each thread block has a private PBSM that is visible only to threads in the same block. The PBSM enables threads within a thread block to cooperate in a fine-grained fashion by sharing data among themselves with low latency; shared memory access is 150 times faster than access to the GPU-main memory.

It has been observed that the matrix multiplications in the numerator in Eq. (2) to compute the cluster covariance matrix are very computationally intensive and therefore a good candidate for GPU acceleration. It was first attempted to use the CUDA built-in kernel function for matrix multiplication, where the multiplication of two matrices is simultaneously performed in multiple areas (thread blocks) of the covariance matrix [10]. However, we found that a simple adaption of the CUDA matrix kernel to Eq. (2) is of limited use to maximize parallel computations because the sum of each multiplication, $\sum_{k=1}^{N} (Z_k - V_i^{(t)}) (Z_k - V_i^{(t)})^T$, requires $N$ iterative sequential additions in each thread. To make these massive iterative statements executable in parallel, Eq. (2) was amended as:

$$M_i = \left(\sum_{k=1}^{N} \mu_{ik}^{(t-1)} \times (Z - V_i^{(t)}) \right) (Z - V_i^{(t)})^T \sum_{k=1}^{N} \mu_{ik}^{(t-1)},$$

where the sum of $N$-additions is removed, and the block-based parallel multiplication is performed on the whole data matrix $(Z)$, instead of for each data point $Z_k$. The $\times$ operator performs element-wise multiplication used in MATLAB. Figure 1 shows the improved parallelization of the matrix multiplication using Eq. (5). Given two matrices $(Z_k - V_i)$ and $(Z_k - V_i)^T$ where their sizes are $[f \times 1]$ and $[1 \times f]$ respectively, $N$ additions of their multiplications for $1 \leq k \leq N$ are parallelized by a single $[f \times N]$ and $[f \times N]$ multiplication.

In addition, to optimize further the usage of shared memory for the matrix multiplication, a parallelization from the viewpoint of physical processors must be considered. In a GPU, each thread block is assigned to a single streaming multiprocessor (SM) configured with massive cores and runs as a unit to completion without preemption. However, under the conventional CUDA kernel for matrix multiplication, all the SMs are not fully exploited when the number of SMs is larger than the required number of blocks for multiplication. For example, consider the case of Fig. 2. Given the feature dimension ($f = 32$) and the block size (16, a fixed default in CUDA), four blocks (SM1, SM2, SM3, SM4) are required for parallel multiplication. In the case of the eight SMs available in a GPU, the remaining four SMs (SM5–SM8) remain idle, which limits the maximum performance obtainable. This problem becomes more worse as the number of features decreases.

To run as many SMs as possible in parallel for a dataset with a small value of $f$, an additional parallelization was implemented over the data point $(N)$. In a typical multiplication in Fig. 2 (left), each block (SM1) sequentially moves
along the \([f \times N]\) matrix and performs the multiplication
\(N'(=N/b\text{locksize})\) times. It was found that the sequential
\(N'\) operations can be parallelized by exploiting the available
idle SMs. In Fig. 2 (right), SM5 is simultaneously active
in the matrix area where SM1 is performing calculations.
The atomic operation in the CUDA kernel was used here to
divide several blocks and to accumulate the partial multiplica-
tion results without interference from any other threads.
Similarly, SM6, SM7, and SM8 perform multiplications in
parallel for each block of the covariance matrix. This matrix
optimization is also applied to the parallel implementation
of the other steps.

3. Experimental Result

To test the effectiveness of the proposed method, the per-
formance of the CUDA-based GK algorithm was compared
with that of the conventional CPU-based GK algorithm. All
experiments were performed on the most cost-effective and
widely used PC with an Intel i5 750 (2.66 GHz, 4 cores,
8 MB cache) CPU with 4 GB main memory and a Geforce
GTX 470 graphic card (607 MHz, 420 cores in 14 streaming
multiprocessors, 1.2 GB of device memory). The CPU used
was a quad-core processor, but only a single core was used
for testing. The software environments was Xcode on Mac
OS X, GCC 4.2 compiler and CUDA SDK v3.2.17. Data
sets were generated using the Gaussian distribution function
in MATLAB. Because the performance is independent of
the distribution, the data were made up of randomly placed
points. To observe the influence of the number of data on
performance, 1K to 128K instance data were created with
various values of \(f\). The comparison results include data
transfer overheads between GPU and CPU.

Figure 3 (a) shows the performance of the CUDA ver-
sion of the GK algorithm over the CPU version as the num-ber of data points increases (\(f = 48\)). The CUDA version
achieves a maximum 240-fold speedup over the single-CPU
version; for clarity, we additionally plotted the speedup of
the single-CPU version when the data transfer overheads
were ignored. For more details, see Fig. 3 (b), which shows
the speedup ratio for each step of the CUDA-based GK algo-

![Fig. 3](image-url)  
**Fig. 3** Performance of the CUDA-based GK for datasets with \(f = 48\).  
(a) Speedup over the single-CPU version for various \(N\).  
(b) Detailed speedup in each step of the CUDA version.

Table 1 lists a summary of the performance of the CUDA
version over the single-CPU version for different numbers
of features; the speedup increases rapidly as the number of
data points and the number of features increase.

To examine the effectiveness of the additional paral-
elization on data point (Fig. 2), the speedup of Step 2 in the
CUDA version was plotted for data sets with \(f = 32, 48, 80\)
in Fig 4. Figure 4 (a) shows the speedup of Step 2 us-
ing Eq. (2); the use of GPU shows better performance than
the single-CPU version. Figure 4 (b) shows the speedup of
Step 2 using Eq. (5), providing much higher speedup than
the Fig. 4 (a); it achieved maximum six-fold speedup. How-
ever, the speedups at \(f = 32\) are less than at \(f = 80\). Fig-
ure 4 (c) shows the increased speedup of Step 2 at \(f = 32, 48\)
when the parallelization in Fig. 2 is used in addition; the
strategy of block rearrangement into idle SMs yielded im-
proved parallelization even for smaller values of \(f\).

Table 2 lists the rates of execution-time for each step of
the single-CPU and GPU versions for three sizes of data set.
For the single-CPU version, Steps 2 and 3 occupy over 95% of
the entire execution time. Moreover, these rates remain
unchanged regardless of the size of the data set. However, in

### Table 1  
Speedup over the single-CPU version for various \(N\) at \(f = 16, 32, 48\).

| \(f\) | \(N\) | 2K | 4K | 8K | 16K | 32K | 64K | 128K |
|------|------|----|----|----|-----|-----|-----|------|
| 16   | 19.0 | 27.8 | 34.2 | 40.4 | 45.7 | 49.4 | 53.6 |
| 32   | 24.5 | 40.6 | 58.7 | 75.0 | 89.5 | 119.3 | 147.3 |
| 48   | 23.7 | 40.8 | 64.6 | 92.1 | 117.6 | 218.1 | 239.5 |

### Table 2  
Percentage comparison of execution time for each step.

| Data(N) | Step 0 | Step 1 | Step 2 | Step 3 | Step 4 |
|---------|--------|--------|--------|--------|--------|
| 2K      | 1.0%   | 1.1%   | 50.9%  | 46.9%  | 0.1%   |
| 16K     | 1.0%   | 1.3%   | 52.9%  | 44.7%  | 0.1%   |
| 128K    | 0.6%   | 1.7%   | 71.1%  | 26.6%  | 0.1%   |
| 2K      | 4.1%   | 5.4%   | 8.1%   | 84.0%  | 0.4%   |
| 16K     | 12.5%  | 12.5%  | 17.8%  | 56.6%  | 0.6%   |
| 128K    | 21.9%  | 17.7%  | 24.5%  | 35.0%  | 0.9%   |
the CUDA version, Steps 2 and 3 take much less execution time than their single-CPU counterparts as the size of the data set increases. This is due to the considerable speedups of these two steps as depicted in Fig. 3(b). By the way, it was observed that the required time for Step 0 in the CUDA version increases due to the need for GPU memory allocation and the need to move data between GPU and CPU.

4. Discussion

It is apparent from the tests that the CUDA version outperformed the single-CPU version by a factor of 200. Moreover, from Fig. 3, we see that a rapid increase in the speedup is obtained when \( N \) is changed from 40,000 to 50,000. Through an in-depth examination, it became clear that the execution time for the single-CPU version increased more rapidly as the number of data points increased beyond \( N = 50,000 \), and that this phenomenon occurs when the total matrix size in bytes exceeds the cache size in the CPU.

Figure 5 (left) shows the execution time of the single-CPU version (8 MB cache size) for calculating the covariance matrix for data point in Step 2. It can be observed that the execution time increases suddenly when the data set size (8 MB, marked by the dotted line) reaches the CPU cache size. From this point, a heavy memory-copy operation from the main memory to the CPU is required. To confirm this argument, the same experiment was performed in another CPU environment using a PC with a Celeron CPU with 1 MB cache. Figure 5 (right) shows the same result as that of Fig. 5 (left); the single-CPU version starts to take much more execution time when the data set size exceeds the CPU cache size (1 MB). However, in the CUDA version, this problem did not arise regardless of the data and cache sizes because the CUDA provides mechanisms for users to control the GPUs memory model directly, such as shared memory management; this is another clear advantage of the CUDA over the single-CPU version.

Despite the remarkable performance of the CUDA version, several issues require further investigation. In Fig. 6, it is apparent that performance degradations in the CUDA version arise at \( f = 17 \) and \( f = 33 \) because the block size is sensitive to the size of the features. A single block is sufficient to deal with a data set with \( f = 16 \), whereas a second block must be allocated if the size of the feature set is 17. However, only a single thread is running in the second block to deal with the seventeenth feature; the remaining \( 16^2 - 1 \) threads in the second block are wasted, which limits the potential to achieve the maximum performance from the GPU. A further study is required to choose the most appropriate block size automatically according to the data size.

Acknowledgements

This research was supported by Basic Science Research through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0004113).

References

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron, “A performance study of general-purpose applications on graphics processors using CUDA,” J. Parallel and Distributed Computing, vol.68, pp.1370–1380, 2008.
[2] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell, “A parallel implementation of K-means clustering on GPUs,” Proc. International Conference on Parallel and Distributed Processing Techniques and Applications, 2008.
[3] B. Hong-tao, H. Li-ia, O. Dan-tong, L. Zhan-shan, and L. He, “K-means on commodity GPUs with CUDA,” World Congress on Computer Science and Information Engineering, pp.651–655, 2009.
[4] M. Zechner and M. Granitzer, “Accelerating K-means on the graphics processor via CUDA,” International Conference on Intensive Applications and Services, pp.7–15, 2009.
[5] J. Espenshade, A. Pangborn, G. von Laszewski, D. Roberts, and J.S. Cavenaugh, “ Accelerating partitional algorithms for flow cytometry on GPUs,” IEEE International Symposium on Parallel and Distributed Processing with Applications, pp.226–233, 2009.
[6] D. Gustafson and W. Kessel, “Fuzzy clustering with a fuzzy covariance matrix,” Proc. IEEE Conference on Decision and Control, pp.761–766, 1979.
[7] D. Kim, K.H. Lee, and D. Lee, “Detecting clusters of different geometrical shapes in microarray gene expression data,” Bioinformatics, vol.21, no.9, pp.1927–1934, 2005.
[8] O. Georgieva and D. Fileve, “Gustafson-kessel algorithm for evolving data stream clustering,” International Conference on Computer Systems and Technologies, pp.14.1–14.16, 2009.
[9] A.S. Raghuvanshi, S. Tiwari, R. Tripathi, and N. Kishor, “GK clustering approach to determine optimal number of clusters for wireless sensor networks,” IEEE Conference on Wireless Communication and Sensor Networks, pp.1–6, 2009.
[10] NVIDIA Corp., CUDA C Programming Guide v3.2, pp.21–28, 2010.