Parallelized k-Means Clustering by Exploiting Instruction Level Parallelism at Low Occupancy

Adhi Prahara, Dewi Pramudi Ismi
Department of Informatics,
Faculty of Industrial Technology,
Universitas Ahmad Dahlan, Indonesia
adhi.prahara@tif.uad.ac.id, dewi.ismi@tif.uad.ac.id

Achmad Imam Kistijantoro, Masayu Leylia Khodra
Department of Informatics,
School of Electrical Engineering and Informatics, ITB
Bandung, Indonesia
imam@stei.itb.ac.id, masayu@stei.itb.ac.id

***Abstract*—Clustering is a technique to cluster data into defined number of cluster. K-means clustering is the most well-known and widely used clustering algorithm. While data become large in terms of volume, the needs of high performance computing (HPC) to perform data clustering is raising. One of the solutions with compromised budget but high efficiency is to utilize highly parallel architecture of Graphics Processing Unit (GPU). In this research, k-means clustering algorithm is implemented on GPU and optimized by exploiting instruction level parallelism (ILP) at low occupancy. ILP on k-means clustering algorithm is achieved by running a number of independent instruction per thread i.e. when calculating distance or sum of data in each cluster. By loading more works into thread at lower occupancy, the higher utilization can be achieved. Experiment on clustering several data shows that the proposed method can speed up k-means clustering several times faster than other parallelized k-means clustering and k-means implementation on CPU.**

Keywords—k-means clustering; parallel computing; instruction level parallelism; low occupancy; CUDA

I. INTRODUCTION

One of the most prominent data analytics tasks is to create groups of data collection based on similarities between data. This grouping activity is called clustering. Clustering activity is usually performed for three main purposes; knowing the underlying structure of the data, natural classification of the data, and compressing the data by taking a prototype from each group of data [1]. Clustering is an unsupervised task. It means that no information/label about the data is known before the data is grouped. Similarity between data is measured from the intrinsic properties held by the data. The number of clusters generated from clustering activity is a predefined input variable. Each generated cluster contains data which are similar to each other. In each generated cluster, sum of distance between data points and its cluster center is minimized.

Clustering algorithms in the literature can be classified into partition clustering, hierarchical clustering, density based clustering and grid based clustering. The most legendary partition clustering algorithm is k-means clustering. K-means clustering is popular due to its easy and simple implementation and its favorable outcome [1].

Researchers have previously conducted attempts to improve performance of k-means clustering algorithms. In general, efforts to improve performance of k-means clustering are performed through two different approaches. The first approach is done by modifying the original algorithm such that computation time can be reduced. Cluster center initialization becomes major concern in this matter since in the original k-means clustering initial cluster centers are appointed in random manner and thus affecting the number of iteration needed to reach convergence. The second approach is done by increasing hardware's computation capacity by employing parallel architecture such as OpenMP, Hadoop MapReduce, and GPU.

In this research we focus on parallel k-means clustering implemented on GPU. There are several existing GPU based k-means clustering, i.e. GPUMiner [2], UV_KMeans [3], HP_KMeans [4], and also [5-9]. With respect to the implementation of parallel k-means clustering in GPU, distance calculation and centroid update are two tasks in k-means which are implemented in parallel fashion. In [5] a specific centroid initialization is also proposed in parallel fashion besides distance calculation and centroid updates. Thread scheduling, solution of finding top-k elements, and parallel high dimension reduction are proposed in [6]. The challenge of GPU based k-means lies on the memory utilization. GPUMiner designed a bitmap to store the nearest centroid of each data point and used corresponding bits to update centroids. Major limitation of GPUMiner is that it uses global memory to store all data points and thus intensive global memory access causes latency. UV_KMeans tried to avoid global memory latency by copying all data points into texture memory and storing centroids in constant memory. UV_KMeans speedup is achieved by using cache mechanism to get high reading efficiency.

According to [10], better performance in GPU computation can be achieved by employing low occupancy. It declaims that hiding latency in GPU is only achieved when occupying more threads. It proves that increasing Instuction Level Parallelism (ILP) is another means of hiding arithmetic latency. Moreover it also shows that using fewer threads, faster computation is resulted as more registers per thread can be occupied.

This work aims at improving the performance of previous implementation parallel k-means clustering in GPU. Although several implementation of GPU based k-means clustering existed, ILP is not employed on those previous implementation. In this work we propose a parallelized k-means clustering by exploiting Instruction Level Parallelism (ILP) at low occupancy to gain better performance/faster execution time. In summary, the contribution of this work is given as follow:

978-1-5386-0658-2/17/$31.00 ©2017 IEEE

a. Calculation of distance between data to each cluster centers and sum of data in each cluster are optimized to gain the benefit of ILP.
b. The kernels are launched using small number of threads to maintain low occupancy and achieve higher utilization by loading more works on thread.

The rest of this paper is organized as Section 2 presents the proposed parallelized k-means clustering, Section 3 presents the results and discussion, and the conclusion of this work is described in Section 4.

II. METHODOLOGY

A. k-Means Clustering

K-means clustering is an algorithm to partition data into k cluster. Data are grouped by the shortest distance between data and each cluster through iterative procedure. If the number of data is N, dimension of data is D, and the number of cluster is K then the general procedure of k-means clustering algorithm can be explained in the following steps.

1. Initialize cluster centers. There are some methods to initialize cluster centers such as using random data, seeding, or simply use the first K data.
2. Find the minimum distance between data to each cluster centers using (1) where d_i is the distance of i^{th} data, x_i is the i^{th} data, and μ_j is the j^{th}cluster center. The distance measurement usually uses Euclidean distance or can be any distance metric.
$$d_i = arg \min_{1 \le j \le K} \|x_i - \mu_j\|^2 \quad (1)$$
3. Assign data to the nearest cluster center from step 2.
4. Compute the new cluster centers using (2) where μ_j is the j^{th} cluster center, n_j is the number of data in j^{th} cluster, and x_i^j is the i^{th}data belong to j^{th} cluster.
$$\mu_j = \frac{\sum_{i=1}^{n_j} x_i^j}{n_j} \quad (2)$$
5. Repeat step 2 to step 4 until converged or reach the termination criteria. There are some convergence terms such as the members of each cluster did not change, the distances between new cluster centers and previous cluster centers are less than threshold, or simply the iteration has reached maximum number of iteration.

Parallelized k-means clustering is done with the same procedure as the original k-means, but the computation is performed in parallel fashion. In this research, authors use the first K data to initialize cluster centers, Euclidean distance as distance measurement, also a distance threshold and maximum number of iteration as termination criteria.

B. Low Occupancy in GPU

GPU is a highly parallel architecture consists of streaming multiprocessors (SM). GPU is structured into grids, blocks, and threads and has several types of memory e.g. global memory, local memory, shared memory, registers, constant memory, and texture memory. Every memory has different latency and the slowest is global memory. According to [11], the access to global memory should be minimized and perform the operation on faster memory such as registers or shared memory.

Another optimization is to hide the latency by running more threads on multiprocessor or on thread block. The Thread Level Parallelism (TLP) can be achieved with this procedure by assigning a number of independent operations on different thread. This procedure also increases the occupancy. However, by maximizing the occupancy sometimes resulted in loss performance [10]. Therefore, authors perform parallelized k-means clustering at low occupancy by using a few numbers of threads in a block in this research.

At lower occupancy, the latency can also be hidden with Instruction Level Parallelism (ILP) beside TLP. ILP can be achieved by performing a number of independent instructions on a thread. This procedure can increase the utilization [10]. The benefits of lower occupancy are no need to synchronize the thread if the number of thread is within the warp and the threads will have more memory resources.

C. Instruction Level Parallelism

ILP is a measure of average number of instructions that can be executed at the same time by a processor. ILP relies on dependency between instructions. Instructions can be executed simultaneously if the instructions are independent to each other, otherwise the instructions must be executed in order. There are different types of dependencies that affect ILP which explained as follow.

1. **Data dependencies**. The dependency occurs when the result of one instruction is used directly or indirectly by another instruction.
2. **Name dependencies**. The dependency occurs when instructions try using the same register or memory location. The common case is one instruction tries to read a memory location while another instruction tries to write on that memory location or some instructions try to write on the same memory location.
3. **Data hazard**. Data hazard occurs when there is data dependency between instructions and close enough to make the pipeline to stall. The common cases are RAW (read after write) when one instruction tries to read a source before another instruction write it, WAW (write after write) when one instruction tries to write a source before it is written by another instruction, or WAR (write after read) when one instruction tries to write a source before another instruction read it.
4. **Control dependencies**. The dependency that determines the order of instructions with respect to branch. Instructions inside a branch cannot be moved before the branch evaluated.

By avoiding the previously mentioned dependencies, ILP can be achieved. In k-means clustering algorithm, the potential operations to be executed in ILP are the calculation of distance and sum of data in each cluster.

III. RESULT AND DISCUSSION

Our proposed parallelized k-means clustering is written in C++ with CUDA and tested on Intel Core i7 6700K, NVidia GeForce GTX 1070 8GB, and 16GB of RAM. Authors use KDD Cup 1999 dataset [12], the same dataset used in [2] which consist of 50,000, 100,000, and 200,000 data with 41 dimensions. Authors analyze the speed up using variation of ILP levels and

number of thread. The experiment compares our proposed parallelized k-means clustering with k-means implementation on CPU and k-means bitmap from GPUMiner [2] by varying number of data and number of cluster.

A. *Parallelized k-Means Clustering*

Our proposed parallelized k-means clustering utilizes two kernels. The first kernel mainly computes two tasks: find minimum distance between data and cluster centers and also sum of data and number of data in each cluster. The second kernel computes new cluster centers. The implementation of our proposed parallel k-means is explained as follow.

1) *Optimization in distance calculation*

The closest Euclidean distance between data and each cluster centers is used to determine the membership of data. To find the closest distance, cluster centers are compared repeatedly as many as the number of data. This condition can make excessive access to global memory. Therefore, authors create shared memory allocation for cluster centers to reduce access to global memory and gain the benefit of faster latency of shared memory.

On distance calculation, the algorithm is optimized for ILP. If D is the number of data dimension and L is the number of ILP level then Algorithm 1.a. shows the distance calculation and Algorithm 1.b shows the distance calculation which optimized for ILP. In Algorithm 1.a., the instructions have data dependencies and will run D times to give the sum result.

In Algorithm 1.b, the distance calculation is divided into three parts. The ILP part (marked with orange color in line 7-8) makes L heavy distance calculations executed simultaneously because the instructions are independent. The sum part (marked with green color in line 10-11) sums the L distance data which saved in static memory. The remainder part (marked with blue color in line 13-14) computes the distance using Algorithm 1.a. The remainder part will only be computed when the number of dimension is not divisible by the number of ILP level.

Algorithm 1.a. Distance calculation

```
dist ← 0
for j = 0 to D do
    dist ← dist + (data[j] − centroid[j])²
dist ← sqrt(dist)
```

Algorithm 1.b. Distance calculation optimized for ILP

```
tD ← D/L                    // divide the number of dimension with ILP level
remD ← mod(D, L)            // remainder after division
temp[L]                     // static memory in registers

dist ← 0
for j = 0 to tD do
// ILP part
    for r = 0 to L do                                              (7)
        temp[r] ← (data[j + r · tD] − centroid[j + r · tD])²       (8)
// sum part
    for r = 0 to L do                                              (10)
        dist = dist + temp[r]                                      (11)
// remainder part
for r = 0 to remD do                                               (13)
    dist ← dist + (data[r + L · tD] − centroid[r + L · tD])²       (14)
dist ← sqrt(dist)
```

2) *Optimization in sum of data calculation*

Before calculating new cluster centers in the second kernel, the first kernel will sum the data and calculate number of data in each cluster. With N data which want to write into K clusters, it can produce data race and excessive access to global memory. Authors use atomic addition in shared memory to solve this problem. Atomic addition ensures that data will be queued if they want to write into the same memory address. The sum of data and number of data in each cluster will be computed partially in shared memory to reduce the data race and global memory access. After the computation is done, each block sums the partially calculated sum of data and number of data in each cluster to global memory using atomic addition.

To further optimize the sum of data computation in Algorithm 2.a., authors use ILP as shown in Algorithm 2.b. Instructions in Algorithm 2.a. are already independent but there is a limit on how many ILP can be executed at once because the limited resources of registers. By using nested loop, the ILP can be maintained. The algorithm is divided into two parts: ILP part (marked by orange color in line 7-8 and in line 16-17) and remainder part (marked by blue color in line 10-11 and in line 19-20). The addition instructions in ILP part are independent and can be executed simultaneously. The ILP level is easily adjusted to lies within the scope memory of registers.

Algorithm 2.a. Computation of partial sum of data to shared memory and the merger to global memory

```
// partial sum of data to shared memory
for j = 0 to D do
    atomicAdd(smem[j], data[j])

... // other routines

// merge partial sum to global memory
for j = 0 to D do
    atomicAdd(sumOfData[j], smem[j])
```

Algorithm 2.b. Computation of partial sum of data and the merger to global memory optimized for ILP

```
tD ← D/L                    // divide the number of dimension with ILP level
remD ← mod(D, L)            // remainder after division
temp[L]                     // static memory in registers

// partial sum of data to shared memory
for j = 0 to tD do
// ILP part
    for r = 0 to L do                                              (7)
        atomicAdd(smem[j + r · tD], data[j + r · tD])              (8)
// remainder part
for r = 0 to remD do                                               (10)
    atomicAdd(smem[r + L · tD], data[r + L · tD])                  (11)

... // other routines

// merge partial sum to global memory
for j = 0 to tD do                                                 (14)
// ILP part
    for r = 0 to L do                                              (16)
        atomicAdd(sumOfData[j + r · tD], smem[j + r · tD])         (17)
// remainder part
for r = 0 to remD do                                               (19)
    atomicAdd(sumOfData[r + L · tD], smem[r + L · tD])             (20)
```

B. *Performance of Parallel k-Means with ILP*

Authors perform experiment to measure the speed up of ILP optimized algorithm in parallelized k-means. The experiment is conducted on dataset of 50,000 data with 41 dimensions and with 24 number of cluster. Fig. 1 shows the computational time according to the variation of ILP level. In Fig. 1, ILP optimized algorithm can decrease the computational time but not in all levels. The drawback using ILP optimized algorithm is when dealing with number of data which not divisible by the number of ILP level, the remainder will be processed in serial manner. Therefore, authors use ILP level two which resulted in smaller remainder of any possibilities number of data.

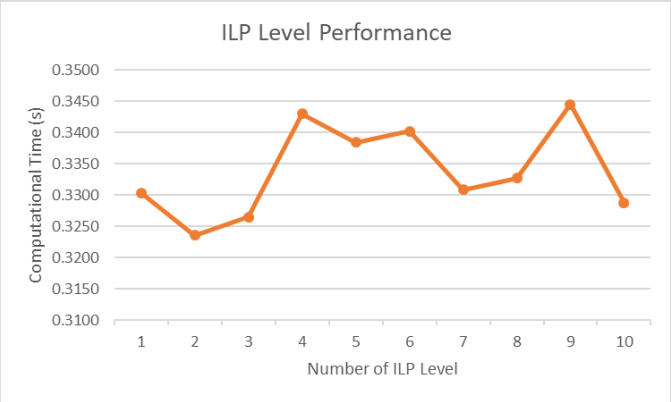

Fig. 1. Performance for different level of ILP

C. *Kernel Execution in Low Occupancy*

The proposed parallelized k-means clustering is executed in low occupancy. According to [10], higher occupancy does not always guarantee better performance. The experiment measures computational time against variation number of thread using dataset of 50,000 data with 41 dimensions and with 24 number of cluster as shown in Fig. 2. From Fig. 2, the performance of the proposed parallelized k-means clustering is better at lower occupancy which has less than 64 threads and worsen when use less than 8 threads. The lowest theoretical occupancy authors can achieve is 18.75% when using number of thread less equal than 32. Authors choose 32 threads to run the kernel to gain the benefit of low occupancy and thread warp. At low occupancy, a thread also has more memory resources and programmer can load more works on thread.

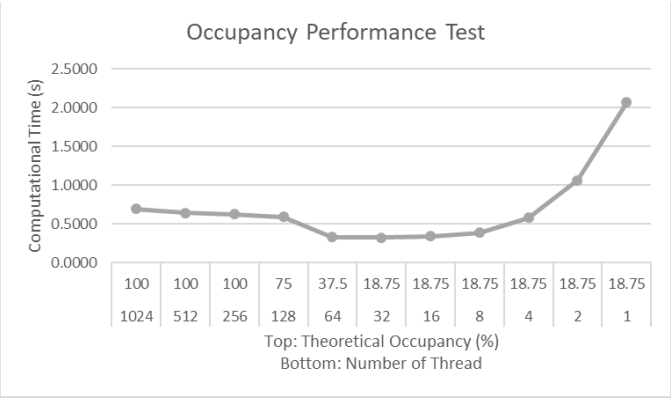

Fig. 2. Performance at different level of occupancy

D. *Performance Comparison*

The performance of proposed parallelized k-means clustering algorithm on GPU is compared with the same implementation on CPU and GPUMiner k-means bitmap [2]. The experiment measures computational time with respect to variation number of data (*N*) and number of cluster (*K*) as shown in Table I. Fig. 3 shows the performance comparison using 50,000 data with 24 clusters. From Table I, the speed-up from CPU implementation is approx. 30x faster on small dataset and approx. 20x faster on large dataset. The comparison of speed up with other GPU implementation is approx. 20 times faster on large dataset. The result proved that our parallelized k-means clustering algorithm with ILP optimized in low occupancy is superior with other implementation on CPU and GPU.

TABLE I. COMPARISON PERFORMANCE BETWEEN DIFFERENT IMPLEMENTATION

K	N	Computational Time (s)		
		CPU	GPUMiner (k-means bitmap) [2]	Proposed Method
24	50,000	10.0950	43.4554	0.3167
24	100,000	34.1760	146.8060	0.9287
24	200,000	64.0850	307.0507	1.9855
50	50,000	30.1850	48.2371	0.9033
50	100,000	87.2750	141.7453	2.6296
50	200,000	165.9790	396.6267	5.1411
100	50,000	45.1780	39.2931	1.9294
100	100,000	169.7930	150.3344	8.2932
100	200,000	388.4790	300.1742	16.2428

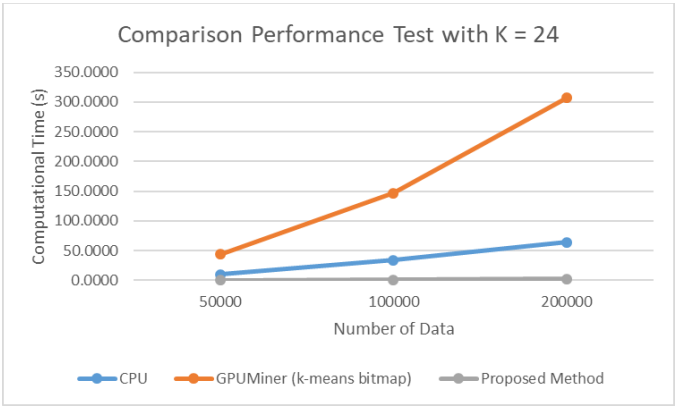

Fig. 3. Comparison performance with 50,000 data

IV. CONCLUSION

The proposed parallelized k-means clustering algorithm exploits ILP at low occupancy. The experiment shows that the proposed method can gain the benefit of ILP level two by reducing the drawback of ILP when dealing with non-divisible number of data with the number of ILP level. At low occupancy with 32 threads in a block, threads can gain more benefit i.e. having more memory resources, thread warps, and by loading more works into thread, higher utilization can be achieved. When compared to CPU implementation and other GPU

implementation (GPUMiner), the proposed parallelized k-means clustering is superior with significant speed up. For future work, authors will use multi GPUs to further increase the performance of parallelized k-means clustering.

ACKNOWLEDGMENT

This research is supported by The Indonesian Ministry of Research, Technology and Higher Education (RISTEK-DIKTI) research grant no. PEKERTI-056/SP3/LPP-UAD/IV/2017

REFERENCES

[1] A.K. Jain, "Data clustering : 50 years beyond K-means", Pattern Recognition Letters, vol.31, pp.651-666, 2010.

[2] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sander, and K. Yang, "Parallel data mining on graphics processors," *Hong Kong Univ. Sci. Techn*

[3] 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 Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, 2008.*ol. Tech. Rep. HKUST-CS08-07*, p. 1, 2008.

[4] R. Wu, B. Zhang, M. Hsu, Clustering billions of data points using GPUs, in: UCHPC–MAW'09: Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop, Ischia, Italy, 2009, pp. 1–6.

[5] J. Bhimani, M. Leeser, and N. Mi, "Accelerating K-Means clustering with parallel implementations and GPU computing," *2015 IEEE High Perform. Extrem. Comput. Conf. HPEC 2015*, 2015.

[6] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, and Y. Shi, "Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)," in *Journal of Supercomputing*, 2013, vol. 64, no. 3, pp. 942–967.

[7] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via CUDA," in *Proceedings of the 1st International Conference on Intensive Applications and Services, INTENSIVE 2009*, 2009, pp. 7–15.

[8] Y. Li, K. Zhao, X. Chu, and J. Liu, "Speeding up k-Means algorithm by GPUs," *J. Comput. Syst. Sci.*, vol. 79, no. 2, pp. 216–229, 2013.

[9] H. T. Bai, L. L. He, D. T. Ouyang, Z. S. Li, and H. Li, "K-means on commodity GPUs with CUDA," *2009 WRI World Congr. Comput. Sci. Inf. Eng. CSIE 2009*, vol. 3, pp. 651–655, 2009

[10] V. Volkov, "Better performance at lower occupancy," in Proceedings of the GPU technology conference, GTC, 2010, vol. 10, p. 16.

[11] S. Cook, CUDA programming : a developer's guide to parallel computing with GPUs. Morgan Kaufmann, 2013.

[12] KDDCup 1999 dataset. Available online at http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. Accessed on September 8th, 2017.