International Journal of Networked and Distributed Computing, Vol. 4, No. 3 (July 2016), 173-181

Paralleled Fast Search and Find of Density Peaks Clustering Algorithm on GPUs with CUDA

Mi Li [1], Jie Huang [2,*], Jingpeng Wang [3]

[1] *School of Software Engineering, Tongji University,*
4800 Cao'An Highway
Shanghai, 201804, China
E-mail: limisky@gmail.com

[2] *School of Software Engineering, Tongji University,*
4800 Cao'An Highway
Shanghai, 201804, China
E-mail: huangjie@tongji.edu.cn

[3] *School of Software Engineering, Tongji University,*
4800 Cao'An Highway
Shanghai, 201804, China
E-mail: jingpeng.wang@outlook.com

Abstract

Fast Search and Find of Density Peaks (FSFDP) is a newly proposed clustering algorithm that has already been successfully applied in many applications. However, this algorithm shows a dissatisfactory performance on large dataset due to the time-consuming calculation of the distance matrix and potentials. In this paper, we proposed a GPU-accelerated FSFDP with CUDA to improve its performance. Thread/block models and the shared memory usage are dedicatedly designed to maximize the utilization of GPUs' hardware resources, and a merge accumulation algorithm based on the odd and even positions of an array is introduced as well. Experimental results show that our parallel implementation of FSFDP can reach a 4.39X and a 15.75X speedup for the calculation of the distance matrix and potentials respectively compared to the serial program on a single CPU core. Higher speedup can be expected for data of larger scales until the device limits are reached. Besides, CUDA stream mechanism is also employed and extra time savings can be obtained by hiding the corresponding memory latency of multiple kernels in a two-way streams' scheduling. Moreover, we evaluate our GPU-based implementation on GPU clusters of 9 nodes and compared to one GPU node, the program can achieve a further 7.55X speedup.

Keywords: Clustering; FSFDP; CUDA; Shared memory; Stream; GPU clusters.

1. Introduction

Clustering attempt to partition elements into clusters on the basis of their similarity. It can find hidden patterns that may exist in datasets and thus plays a critical role in a wide range of applications. Several different clustering strategies have been proposed and each algorithm has its own strength and weakness due to the complexity of information.[1] Recently, a new clustering method called Fast Search and Find of Density Peaks (FSFDP) was proposed on Science in 2014.[2] It has already been successfully applied in estimating ages of facial image[3] ,

* Corresponding author.

text categorization[4,5], image segmentation[6], community detection[7] and discovering demand hot spots in city-scale taxi fleet dataset.[8]

Despite the effectiveness of FSFDP, this method still has some limits. The result of FSFDP is very sensitive to the choice of parameter d_c, which is the distance threshold used to calculate density. Ref. 9 proposed a way to extract the threshold value of d_c from the original dataset by using the potentials' entropy of the dataset. They pointed out that suitable d_c corresponds to the smallest entropy of potentials. However, the calculation amount of finding the smallest entropy is huge due to the multiple computations of potentials for different impact factor σ. In addition, distance matrix is the base of density calculation whose computation time cost is also large, especially when the dataset is huge and with high dimension. With the explosion of information, algorithms with high computation cost can hardly meet the demands of large-scale data processing. Therefore, it is necessary to speed up FSFDP to extend the application scenarios of this newly proposed method.

Traditionally, multi-core CPUs are used to reduce the computation cost of programs. The task is always divided into pieces and each CPU deal with only parts of the task in parallel. However, there are mainly two drawbacks. First, the effect of multi-core CPUs depends on the number of cores and their clock frequency, and it is usually expensive to ensure good performance. Secondly, the CPU clusters are also quite energy-consuming.

From the perspective of both performance and efficiency, we introduce the CUDA technologies to accelerate the FSFDP algorithm on GPUs. GPUs are dedicated hardware for manipulating computer graphics and have evolved into highly parallel many-core processors due to the huge computation demand of 3D graphics. With the help of CUDA platform, general-purpose computing can also benefit from the computing power of GPUs through a C-like programming interface. High performance can usually be gained with a low cost of energy.

In fact, GPU technology, including OpenCL and CUDA platform, has contributed a lot in massive calculations and elaborate algorithms, especially in scientific computing area. Bai, T. et al.[10] use three strategies to accelerate Nth-degree truncated polynomial ring with GPUs and achieve high improvement on performance. Hung, C. et al.[11] develop two efficient GPGPU-based parallel packet classification approaches to filter packets by leveraging thousands of threads.

In this paper, we also use CUDA technologies to accelerate the FSFDP algorithm. Our work mainly focus on the two most compute-intensive portions of this algorithm: 1) the calculation of distance matrix and 2) the computation of potentials for the determination of the parameter d_c. After that, stream scheduling is utilized to encapsulate multiple kernels and hide their corresponding memory latency. Finally, we use GPU clusters with socket communication mechanism to reach a further improvement of performance

The rest of this paper is organized as follows. We first introduce the FSFDP algorithm in section two. Section three discusses the parallelization strategies and section four evaluates the performance of our implementation. Finally, conclusions and future works are presented in section five.

2. Fast Search and Find of Density Peak Clustering

FSFDP algorithm is based on the assumptions that cluster centers usually have two characters:

- with large local density: the number of points within a certain threshold of scope should be large,
- have relatively large distance from any points with a higher local density: cluster centers must be away from other data points that might be the cluster centers.

Based on these assumptions, we will first determine the threshold d_c by calculating the potential of each data point and the potentials' entropy. With d_c selected, we need to compute two quantities for each data point i: its local density ρ_i and its distance δ_i from points of higher density. Both these quantities depend on the distances between data points. Then the cluster centers are picked out based on the decision graph and the remaining data points are put into the nearest cluster with higher local density.

Therefore, the process of FSFDP can be divided into five steps:

(i) Compute the distance matrix.
(ii) Choose parameter d_c.
(iii) Calculate the local density.
(iv) Retrieve the minimum distances from points of higher density.
(v) Clustering.

2.1. Distance matrix

Considering a dataset $S = \{x_i\}_{i=1}^{N}$ and its corresponding indexes $I_S = \{1, 2, ..., N\}$, the elements in the distance matrix can be denoted as $d_{ij} = dist(x_i, x_j)$. There are many different definitions of "distance", while Euclidean distance and Manhattan distance are the most widely used ones. Here we use the Euclidean distance as an example, while the same parallelization can be easily extended to other distance definitions.

Assuming that each data x_i in the dataset S has the dimension of M, the components of data points on each dimension can be written as $\{x_{ik}\}_{k=1}^{M}$. The distance matrix can be calculated by the equation below:

$$dist^2(i,j) = \sum_{k=1}^{M} (x_{ik} - x_{jk})^2. \quad (1)$$

2.2. Choice of threshold d_c

The local density depends heavily on the choice of threshold d_c. Suitable d_c is vital to the effectiveness of the whole algorithm.

We utilized the potential of every point[12] to determine the suitable value of d_c. The formula to calculate the potential of each point can be represented as:

$$\varphi_i = \sum_{j=1}^{N} e^{-\left(\frac{d_{ij}}{\sigma}\right)^2}. \quad (2)$$

where σ is the impact factor.

Ref. 9 pointed out that, impact factor σ should be optimized to ensure the smallest uncertainty of the dataset's potentials, and the uncertainty is usually represented by entropy H:

$$H = -\sum_{i=1}^{N} \frac{\varphi_i}{Z} \log\left(\frac{\varphi_i}{Z}\right).$$

$$Z = \sum_{i=1}^{N} \varphi_i. \quad (3)$$

where Z is the normalization factor.

Suppose that σ^* lead to the smallest value of entropy H, then d_c should be set according to the formula[9] below:

$$d_c = \frac{3}{\sqrt{2}} \sigma^*. \quad (4)$$

2.3. Local density

Given a distance threshold d_c, the local density of point x_i should represent the number of points whose distance from x_i is less than the threshold d_c. Here a Gaussian kernel is used to ensure the continuity of the local density. Therefore, the local density of each point x_i can be defined as:

$$\rho_i = \sum_{j \in I_S \setminus \{i\}} e^{-\left(\frac{d_{ij}}{d_c}\right)^2}. \quad (5)$$

2.4. Minimum distance from points of higher density

Let $\{q_i\}_{i=1}^{N}$ represent the index of sorted ρ_i in descend order, which satisfies that

$$\rho_{q_1} \geq \rho_{q_2} \geq \ldots \geq \rho_{q_N}. \quad (6)$$

The distance from points of higher density δ_i can be defined as:

$$\delta_{q_i} = \begin{cases} \min\left\{d_{q_i q_j}\right\}, i \geq 2 \\ \max\left\{\delta_{q_j}\right\}, i = 1 \end{cases}. \quad (7)$$

That is to say, when the point x_i has the largest local density all over the dataset, δ_i is set to the maximum of all the other δ values. Otherwise, δ_i denotes the distance between x_i and the nearest point who has higher local density than x_i. The index of the nearest point with higher density is marked as $\{n_i\}_{i=1}^{N}$ while n_i can be taken as -1 for the point that has the highest density.

2.5. Clustering

Based on the definition of local density ρ and minimum distance from points of higher density δ, the cluster centers should have relatively large ρ and δ simultaneously. The decision graph[2] composed of ρ and δ can be utilized to choose cluster centers in size of nc. Let $\{y_k\}_{k=1}^{nc}$ be the cluster centers , the cluster index $\{c_i\}_{i=1}^{N}$ can be initialized to:

$$c_i = \begin{cases} k, if\ x_i\ is\ the\ cluster\ center\ whose\ index\ is\ k \\ -1, otherwise \end{cases}. \quad (8)$$

Afterwards, other points can be categorized into the same cluster as their nearest point with higher density. This process can be written as

$$for\ all\ c_{q_i} = -1, c_{q_i} = c_{n_{q_i}}. \quad (9)$$

Finally, the clustering process is completed and the cluster index of each point in the dataset is stored in $\{c_i\}_{i=1}^{N}$.

3. Parallelization and Optimization with CUDA

The most time consuming parts of FSFDP are the calculations of the distance matrix and the determination of d_c due to the multiple computations of potentials for different impact factor σ. In this section, we adopt JCuda[13] wrapper as the host part of the whole program. We will first introduce the thread/block model and the shared memory usage applied in the calculation of the distance matrix and potentials with CUDA.[14] Afterwards, we adopt CUDA stream scheduling mechanism to hide the memory latency in multiple kernels, which will be helpful in computing partial distance matrix on a very large scale dataset and will accelerate the multiple calculations of potentials during the determination of threshold d_c. Besides, we setup GPU clusters of 9 nodes with socket communication model for a further improvement on performance.

3.1. Calculate distance matrix with CUDA

We let each thread responsible for one entry of the distance matrix, and thus N^2 threads are allocated to calculate the distance matrix at the same time. These threads are organized by blocks, and threads in the same block can collaborate on the same segment of shared memory. Here the block size is set to 32×32 because the maximum number of threads in a modern GPU's block is 1024, and the grid size is thus $(N/32)\times(N/32)$. In addition, for N that is not the multiple of 32, we can easily fulfill the rest of the matrix with zeros.

However, the size of data needed for each block is $32\times32\times M\times2$ (each thread need the pair of data points with the size of $M\times2$ and one block consists of 32×32 threads) and this size may exceed the maximum storage of shared memory for a large M. Therefore, the calculation of 32×32 pairs of distances in one block should be resolved into smaller pieces.

According to the definition of distance matrix, it can be found that each entry is the sum of M components and all components are totally independent. Therefore, each block can deal with one component of the distance

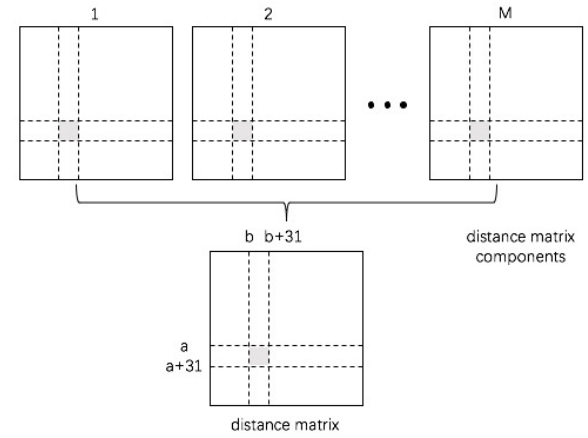

Fig. 1. The sketch of the parallelization strategy on GPU for calculating the distance matrix.

matrix at a time and the data size of shared memory needed in each block is only $32\times32\times2$. The final distance matrix can be retrieved by the accumulation of M components.

This method is represented explicitly in Figure 1 where the grey square shadow in the distance matrix indicates the result of one block with 32×32 threads. Suppose that the grey shadow is the sub distance matrix between points x_a to x_{a+31} and x_b to x_{b+31}, the values in it are the sum of M components. For the m-th component (here m is the dimension index of the point), the data stored in the shared memory are the independent assortment result between the following two sets:

$$\{x_{a,m}, x_{a+1,m}, \ldots, x_{a+31,m}\}, \{x_{b,m}, x_{b+1,m}, \ldots, x_{b+31,m}\}$$

Therefore, the m-th component of the sub distance matrix is:

$$dist_m(x_i, x_j) = (x_{im} - x_{jm})^2. \quad (10)$$

which is represented by the grey shadows in the distance matrix components and the sub distance matrix is thus calculated by the formula below:

$$dist^2(x_i, x_j) = \sum_{k=1}^{M} dist_m(x_i, x_j). \quad (11)$$

3.2. Calculate potentials with CUDA

We allocate 1000 threads and N blocks for the calculation of potentials. Similarly, zeros are fulfilled if needed. Each block is responsible for the computation

of the potential for one point, which is the accumulated sum of N items. Therefore, 1000 threads should collaborate to calculate the N items and their accumulated sum.

For each point x_i, the N distance values $\{d_{ij}\}_{i=1}^{N}$ will first be copied to the shared memory while each thread takes over $N/1000$ portion of the copy operation. Afterwards, each thread calculates

$$e^{-\left(\frac{d_{ij}}{\sigma}\right)^2}$$

for $N/1000$ times, and the sum of $N/1000$ results is stored as one entry into an array in of size 1000. Based on this array, the potential of a certain point can be calculated by the accumulation of each entry in the array. Then the residue work is simplified to compute the sum of an array in size of 1000.

Here we introduce a merge accumulated sum algorithm based on the odd and even position of the array. In detail, all the values on the odd positions of the array are added on the value at its next even position, and the values on the even positions form a new array. This process is iterated until all the values are added to the first element of the array, and finally the value of the first element is the result we want.

In addition, the same parallel strategy can be also applied in the similar calculation of local density (see Eq. (2) and Eq. (5)).

3.3. Distance matrices and potentials with Stream Scheduling

CUDA provides a programming model called CUDA stream. With stream scheduling mechanism, the GPU program schedules multiple kernels simultaneously. One CUDA stream can encapsulate multiple kernels, and they have to be scheduled strictly following a particular order. However, kernels from multiple streams can be scheduled to run concurrently. The main purpose of applying CUDA streams is to hide the memory latency: when kernel A is loading/writing data, kernel B can occupy the cores for computation. As a result, the cores in the multiprocessor can reach better utilization.

We use the streams mechanism to optimize the kernels in the computation of the distance matrices and potentials. Firstly, we use NVIDIA Visual Profiler to analyze the performance of our program, and detail profiles are shown in Figure 2 and Figure 3.

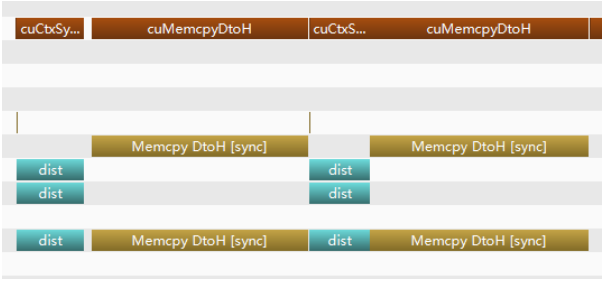

Fig. 2. Performance profile of serial kernels on calculating distance matrices from NVIDIA Visual Profiler.

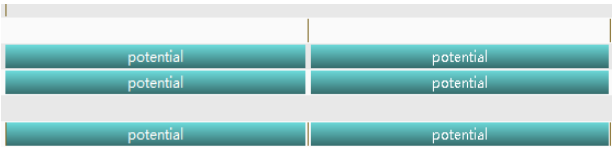

Fig. 3. Performance profile of serial kernels on calculating potentials from NVIDIA Visual Profiler.

From the profiles, we can obviously see that there exist memory latency during the process of data transmission from device to host, especially heavy in the calculation of the distance matrix. Besides, I/O time cost from device to host is longer than that of kernel execution. Therefore, stream scheduling is very necessary to improve the performance of the program with multiple kernels.

We allocate a buffer on the host side and mark it as pinned memory to store the resulted distance matrix from GPU temporarily by Java NIO[15] APIs. In this way, GPU device can communicate with the host like Direct Memory Access (DMA). When one stream is executing kernels, the other one can write data to the allocated buffer. Similarly, we also use this way to implement stream scheduling for potential computing. The detail pseudo code for distance matrices computing is as follows:

Pseudo Code: Stream Scheduling Optimization for Distance Matrices

Input: Target point matrix *devicePoints*;
the row of the point matrix *N*;
the column of the point matrix *M*.
Output: Multiple distance matrices *dist.*
Begin:
STREAM_NUMBER = 2;
prepare compiled PTX file

initialize *CUdevice*, *CUcontext*, *CUmodule*, *CUfunction*
create 2 *CUstreams* as an array variable *streams*
cudaHostAlloc a byte buffer
cuMemAlloc input variables :
devicePoints, *N*, *M*
for i = 1 to STREAM_NUMBER **do**
    cuMemcpyHtoDAsync (*streams*[i % 2])
end
for i = 1 to STREAM_NUMBER **do**
    cuLaunchKernel (*streams*[i % 2])
end
for i = 1 to STREAM_NUMBER **do**
    cuMemcpyDtoHAsync (ByteOffset, *streams*[i % 2])
end
read NIO allocated buffer.
End

3.4. Distance matrices and potentials with GPU clusters

Furtherly, based on previous optimized GPU kernel, we implement the algorithm to compute multiple distance matrices and potentials on our GPU clusters to accelerate the calculation speed of the FSFDP algorithm. We adopt socket with customized data packet to handle the communication between GPU nodes. The data packet consists of corresponding input data. Detail architecture of the communication model among different nodes can be seen in Figure 4. A node of the clusters is set as the master node, and other 8 nodes are responsible for the communication to it as slave nodes.

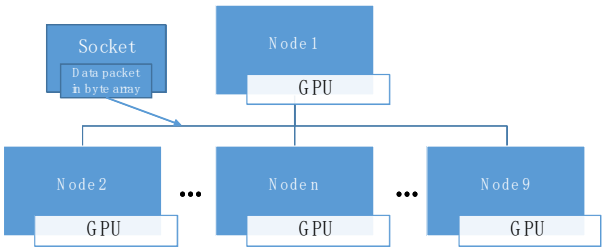

Fig. 4. Communication model of GPU clusters.

4. Experimental Results

4.1. Experimental environment

A dataset called dim-sets[16] is used to compare the performance of CPU-based and GPU-based FSFDP. This dataset contains synthetic data with Gaussian clusters in multi-dimensional space. The summary of the dataset can be found in Table 1.

We performed all of our experiments on Intel(R) Core(TM) i7 4790 CPU and NVIDIA GTX 970 GPU. The CPU has eight cores running at 3.6GHz and the memory size is 16GB. GTX 970 has its frequency at 1.29GHz and the size of its display memory is 4096M.

The experiments contain four parts. Firstly, we compare the time consumption of calculating the distance matrix between CPU and GPU programs. Then, the performance of calculating the potentials is also compared. Meanwhile, we observed how much the performance can benefit from the stream scheduling on both parts. Finally, we conducted our program on GPU clusters to evaluate the extensibility of our parallelization.

Table 1. Summary of the Dim-sets dataset.

Dataset name	Number of data vectors (N)	Dimension of data vector (M)
dim3	2026	3
dim6	4051	6
dim9	6076	9
dim12	8101	12
dim15	10126	15

In addition, all the time costs have been measured for 10 times, and the average results are reported. Besides, the time cost on the creation of GPU context and memory release are both taken into account when measuring the GPU program's performance.

4.2. Performance of calculating distance matrix

We first analyze the performance of calculating the distance matrix. From Figure 5 we can see that, for datasets dim3 and dim6, the time cost of the GPU paralleled program is higher than that of the serial CPU program. This is because the number of data is quite

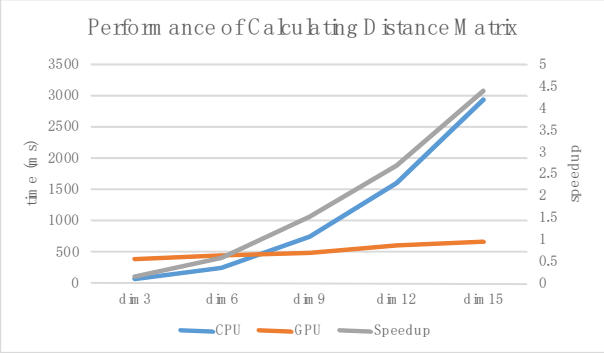

Fig. 5. Performance of calculating the distance matrix.

small, and time cost on launching CUDA context takes up a larger proportion than the kernel and I/O.

However, for the following datasets when both the number and dimension of the data vector increase, we can obviously see that the time cost on the serial CPU program arise dramatically while the increment of that on GPU is few. For dataset dim15, the time cost of calculating the distance matrix with a single CPU core is 2940ms while GPU only takes 669ms. The speedup is presented by the grey line in the figure, and it clearly shows an increasing trend with the expansion of data scale. As for the largest dataset dim15 in our experiments, the speedup is 4.39X. Moreover, we can infer that the speed-up ratio will be higher on larger datasets until the device limits are reached.

4.3. Performance of calculating potentials

The performance of calculating the potentials is shown in Figure 6. Comparing with the performance of calculating the distance matrix, the time cost has a similar trend and the speedup is better than that for the distance matrix. For dim15, the time cost of CPU is

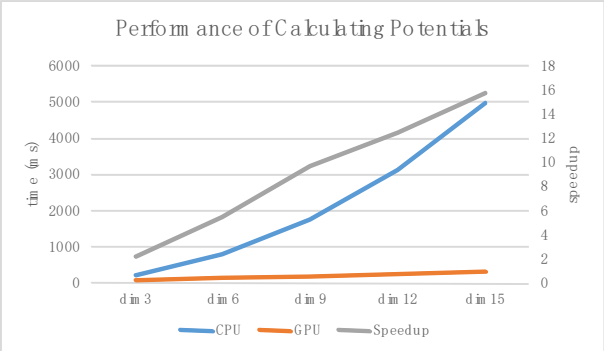

Fig. 6. Performance of calculating the potentials.

4964ms and 315ms for GPU, the speedup is as high as 15.75X. Still the device resource limit has not been reached and better speedup can be expected for dataset of larger scales.

4.4. Performance of stream scheduling

Considering that the result from device to host in computing the distance matrix is a large matrix with size in $N \times N$, and the time cost on its I/O is longer than that on GPU kernel, we apply CUDA stream scheduling mechanism to optimize our program and implement kernel concurrency.

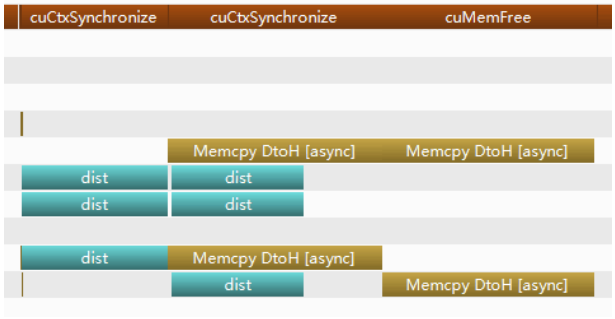

Fig. 7. The profile of stream scheduling on calculating the distance matrices.

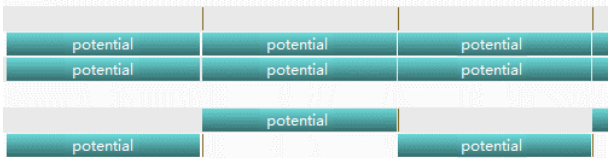

Fig. 8. The profile of stream scheduling on calculating the potentials.

Firstly, we evaluate the difference of the time cost on I/O and GPU kernel. The profile retrieved from Visual Profiler indicates that, for a given (N, M) = (16000, 32), the duration of kernel is 57.41ms on average while that of memory copy from device to host is 84.346ms. Therefore, we register a pinned memory buffer to complete the copy operation on 2 streams alternately, when one stream is invoking the kernel function, the other one is dealing with a memory copy. In this way, we implement a two-way stream scheduling like a pipeline.

dist	
Start	2.109 s (2,109,464,277 ns)
End	2.167 s (2,166,873,858 ns)
Duration	57.41 ms (57,409,581 ns)
Grid Size	[500,500,1]
Block Size	[32,32,1]
Registers/Thread	26
Shared Memory/Block	8 KiB
Occupancy	
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Fig. 9. The GPU kernel properties on calculating the distance matrices.

rho	
Start	16.806 s (16,806,160,755 ns)
End	16.826 s (16,826,107,776 ns)
Duration	19.947 ms (19,947,021 ns)
Grid Size	[16000,1,1]
Block Size	[1000,1,1]
Registers/Thread	21
Shared Memory/Block	3.906 KiB
Occupancy	
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Fig. 10. The GPU kernel properties on calculating the potentials.

Memcpy DtoH [async]	
Start	2.252 s (2,251,719,455 ns)
End	2.335 s (2,335,310,788 ns)
Duration	83.591 ms (83,591,333 ns)
Size	1.024 GB
Throughput	12.25 GB/s
Memory Type	
Source	Device
Destination	Pinned

Fig. 11. The Memcpy DtoH properties on calculating the distance matrices.

Memcpy DtoH [async]	
Start	16.785 s (16,785,189,559 ns)
End	16.785 s (16,785,219,096 ns)
Duration	29.537 µs
Size	64 kB
Throughput	2.167 GB/s
Memory Type	
Source	Device
Destination	Pinned

Fig. 12. The Memcpy DtoH properties on calculating the potentials.

From Figure 7 we can find that the optimized kernel with two-way stream scheduling can save $(X-1)\times T_{kernel}$ in time than serial kernels, where T_{kernel} is the time cost of executing kernel function on average, and X is the computing times of the kernel. Similarly, we also apply stream scheduling on the calculation of potentials. However, when calculating the potentials, the result returned only has the size of N. Thus, the time cost of kernel is much longer than the time cost on memory copy from device to host, the effect of stream scheduling on this case is not very ideal. But according to Figure 8, the two-way streams can still save $X\times T_{I/O}$ in time, where $T_{I/O}$ is the time cost of I/O on average, and X is the invoking times of the kernel function.

Here we also plot the GPU kernel properties in Figure 9 and Figure 10 from the result we can see the detail thread/block configuration is that: for distance matrices computing, the Block Size is [32, 32, 1] and the Grid Size is [500, 500, 1]. And at the same time, we can also find the theoretical occupancy has already reached 100%. That means the GPU device has already been fully utilized. Similarly, for potentials computing, the Block Size is [16000, 1, 1] and the Grid Size is [1000, 1, 1]. We can also find the theoretical occupancy is 100%.

Besides the kernel properties, we also investigate the memory copy operation (from device to host) and find out that the size of memory copy (from device to host) in computing distance matrices is about 1.024GB, the time cost on take a larger proportion on the total executing time while in computing potentials, the size on memory copy is only 64KB and the time cost on memory copy is about 30µs. Therefore, it is obviously that stream scheduling on computing distance matrices can achieve much better speedup than that on computing potentials. Through result analysis, stream

scheduling on calculating multiple distance matrices (Grid Size is [500, 500, 1], Block Size is [32, 32, 1]) can achieve 1.68 speedup over original GPU kernel while on calculating multiple potentials, the speedup is only about 1.0007.

4.5. Performance of GPU clusters

In our cluster experiment, the GPU kernels executed are the optimized version with CUDA shared memory and stream scheduling. We collect experimental results and achieve a 7.55X speedup when computing 1000 distance matrices and potentials on a cluster of 9 nodes on average over a single GPU node. The result indicates that compare to the theory speedup 9X, the cost on communication between nodes does exist, although the cost on communication is not heavy.

5. Conclusion and Future Work

In this paper, we paralleled the FSFDP cluttering algorithm on GPUs with CUDA. The calculation of the distance matrix and potentials are focused because they are the most time-consuming portions of the algorithm. With our parallel strategies, the time cost of calculating the distance matrix can reach a speedup at 4.39X and the computation of potentials obtain a speedup as high as 15.75X compared to the serial program on a single CPU core. Higher speedup can be expected for data of larger scales until the device limits are reached. After that, CUDA stream scheduling mechanism is utilized and $(X-1)\times T_{kernel}$ and $X\times T_{I/O}$ extra time can be saved for X times of invoking the kernels respectively. Moreover, a further 7.55X speedup can be obtained through a 9 node GPU cluster. Our paralleled implementation of FSFDP improves the performance of this algorithm to a large extent which makes it adapt to more application scenarios.

Future work includes but not limit to: design strategies that can find the smallest entropy of potentials and determine the threshold d_c automatically. Besides, we will also try to implement the algorithm on other heterogeneous systems such as IBM POWER 8 with Field Programmable Gate Array (FPGA) and CAPI interface. Use some container technology such as LXC, docker to combine with GPU arrays to accelerate the algorithm.

References

1. D. Xu and Y. Tian, A Comprehensive Survey of Clustering Algorithms, *Annals of Data Science*, **2**(2) (2015), pp. 165-193.
2. A. Rodriguez and A. Laio, Clustering by fast search and find of density peaks, *Science*, **344**(6191) (2014), pp. 1492-1496.
3. Y. Chen, D. Lai, H. Qi, J. Wang and J. Du, A new method to estimate ages of facial image for large database, *Multimedia Tools and Applications*, (2015), pp. 1-19.
4. Y. Chen and J. Du, A new method for classifying chinese text based on semantic topics and desity peaks, *Int J Appl Math Mach Learn*, **1**(1) (2014), pp. 35-54.
5. P. Wang, J. Xu, B. Xu, C.-L. Liu, H. Zhang, F. Wang, et al, Semantic Clustering and Convolutional Neural Network for Short Text Categorization, in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, **2** (2015), pp. 352-357.
6. Z. Chen, Z. Qi, F. Meng, L. Cui and Y. Shi, Image Segmentation via Improving Clustering Algorithms with Density and Distance, *Procedia Computer Science*, **55** (2015), pp. 1015-1022.
7. Y. Li, C. Jia and J. Yu, A parameter-free community detection method based on centrality and dispersion of nodes in complex networks, *Physica A: Statistical Mechanics and its Applications*, **438** (2015), pp. 321-334.
8. D. Liu, S.-F. Cheng, and Y. Yang, Density Peaks Clustering Approach for Discovering Demand Hot Spots in City-scale Taxi Fleet Dataset, in *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*, (2015), pp. 1831-1836.
9. S. Wang, D. Wang, C. Li, and Y. Li, "Comment on" Clustering by fast search and find of density peaks"." *arXiv preprint arXiv:1501.04267* (2015).
10. T. Bai, S. Davis, J. Li, Y. Gu and H. Jiang, Accelerating NTRU Encryption with Graphics Processing Units, *International Journal of Networked and Distributed Computing*, **2**(4) (2014), pp. 250-158.
11. C. Hung and S. Guo, Fast Parallel Network Packet Filter System based on CUDA, *International Journal of Networked and Distributed Computing,* **2**(4) (2014), pp. 198-210.
12. D. Li, S. Wang, W. Gan, and D. Li, Data field for hierarchical clustering, *Developments in Data Extraction, Management, and Analysis*, (2012), p. 303.
13. Jcuda.org, *jcuda.org - Java bindings for CUDA*, (2016), [online] Available at: http://jcuda.org/ [Accessed 5 Apr. 2016].
14. NVIDIA Developer, *CUDA Zone*, (2015), [online] Available at: http://developer.nvidia.com/cuda-zone [Accessed 5 Apr. 2016].
15. Docs.oracle.com, *Home: Java Platform, Standard Edition (Java SE) 8 Release 8*, (2016), [online] Available at: https://docs.oracle.com/javase/8/ [Accessed 9 Apr. 2016].
16. Cs.joensuu.fi, *Clustering datasets*, (2016), [online] Available at: http://cs.joensuu.fi/sipu/datasets [Accessed 5 Apr. 2016].