Parallel Spectral Graph Partitioning on CUDA

Orhan Fırat and Alptekin Temizel

***Abstract*—Parallelization of scientific problems is a challenging task which has a wide application area both on distributed programming, cloud computing and recently on GPGPU. Spectral graph partitioning is a widely used technique in many fields such as image processing, scientific computing, machine learning etc. In this study we analyze spectral graph partitioning subroutines on a GPGPU framework. Each step is analyzed with differing techniques to lead a conclusion about usage of GPGPU on overall spectral graph partitioning algorithms.**

I. Introduction

Unsupervised learning is one of the most important approaches in machine learning where there is no labeled data to train a classifier. There are many reasons and advantages to employ unsupervised learning methods in classification tasks. First, collecting and labeling a large set of sample patterns can be costly. Second, it may be valuable to gain some insight into the nature or structure of the data and unsupervised learning helps this problem by not having any assumption on the distribution of the data.

There are many approaches and techniques for clustering the data in an unsupervised manner. Recently, spectral clustering methods which exploit pair-wise similarities, shown to be more effective in finding clusters than some traditional algorithms such as k-means, fuzzy c-means etc. Spectral clustering is a graph-theoretic clustering algorithm which finds an optimal graph-cut [1] [2]. It is known that graph-cut is NP-hard and spectral clustering solves graph-cut problem by approximation, namely approximating the problem to spectral graph partitioning.

Despite various advantages such as ability to cluster non-Gaussian and arbitrary-shaped clusters, when the data instances n is large, spectral clustering encounters quadratic data bottleneck by computing and storing the pair-wise similarity-matrix. Also spectral clustering requires remarkable time and memory to find and store the first k eigenvectors of the *Laplacian* matrix where k is the number of clusters. Hence it is obvious that spectral clustering algorithm can perform faster by employing parallel approaches.

All experiments and implementations of this study is conducted in the Middle East Technical University – Informatics Institute Vision-Lab.

Orhan Fırat is with the Department of Computer Engineering, Middle East Technical University (e-mail: orhan.firat @ ceng.metu.edu.tr).

Alptekin Temizel is with the Informatics Institute, Middle East Technical University (e-mail: atemizel @ ii.metu.edu.tr).

During recent years, GPGPU has been developing rapidly. Since NVIDIA released CUDA, it has appealed interests of many researchers in all kinds of domains. Implementation of the complete spectral clustering algorithm on GPGPU, though is not a well studied topic so far because of its hard coupling with eigen-value decomposition step. Except that the initial and sequent steps are not only can be considered as a series of sparse vector-vector multiplications but also, spectral clustering algorithm mainly consists of matrix and vector operations. Matrix problems usually imply huge possibility to parallelization so we can come to a conclusion that spectral clustering is well-suited to be parallelized, thus motivated our study to analyze algorithmic subroutines of the clustering algorithm on a GPGPU.

II. Recent Studies

Parallelization of linear systems is well studied in the domain of parallelization, but yet the parallelization of spectral clustering is not a hot topic by consideration of GPGPU. Few studies are found that employ GPGPU in spectral clustering [3]. However, a very important part of the spectral clustering algorithm is the eigenvalue problem of symmetric matrix, is studied by many researchers. In [4] a parallel ARPACK algorithm is employed to perform parallel eigenvalue decomposition and distributed architecture is exploited, also this algorithm gets fast when the matrix is sparse. This algorithm employs parallelism by using distributed systems not GPGPU but one of the examples of parallel solution of spectral clustering.

Another popular approach to speed up spectral clustering is by using a dense sub-matrix of similarity matrix. In another work Chen et al. [5] propose a method by comparing both sparsifying the similarity matrix and using the Nyström approximation which avoids calculating the whole similarity matrix. Their Parallel Spectral Clustering algorithm provides a systematic solution for handling challenges from calculating the similarity matrix to efficiently finding eigenvectors. Also this study does not employ GPGPU in their parallelization algorithm.

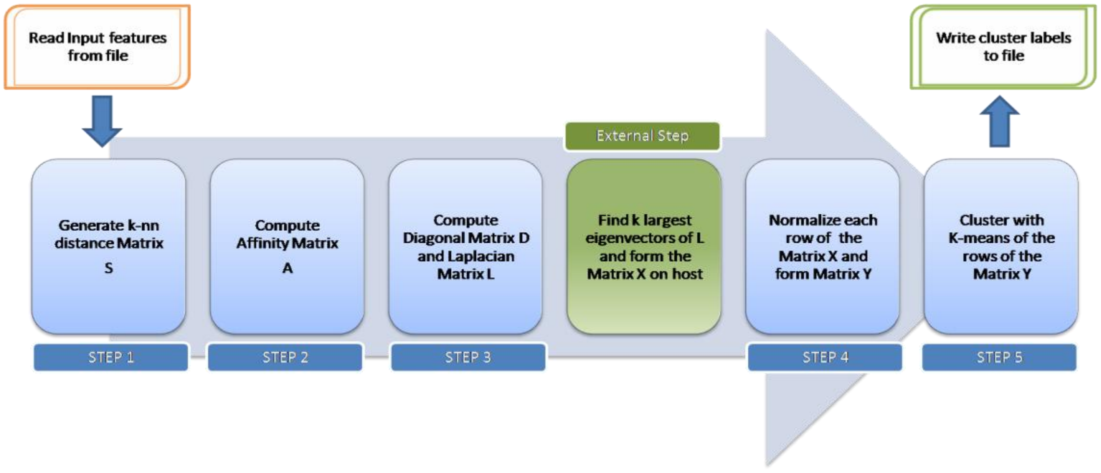
Read Input features from file
Write cluster labels to file
External Step
Generate k-nn distance Matrix S
Compute Affinity Matrix A
Compute Diagonal Matrix D and Laplacian Matrix L
Find k largest eigenvectors of L and form the Matrix X on host
Normalize each row of the Matrix X and form Matrix Y
Cluster with K-means of the rows of the Matrix Y
STEP 1
STEP 2
STEP 3
STEP 4
STEP 5

Figure 1. Flow-chart of the proposed spectral clustering algorithm on GPU. Each module is implemented on GPGPU represented as a separate step with blue boxes. Steps executed on host tier illustrated as green boxes and excluded in analysis.

There exist no more than a few studies that employ GPGPU in order to execute parallel spectral clustering algorithm. The first study that employs GPU in the problem is [3] by Zheng et al. In their spectral clustering framework parallelizing spectral is conducted by first dividing the algorithm into three discrete steps and then each sub-problem solved on GPU sequentially. First affinity matrix construction from similarity matrix is parallelized; second spectral computations are parallelized namely, computing *Laplacian* matrix and its eigenvectors corresponded to the first k largest eigenvalues of L by parallelization of dense matrix-vector multiplication. Finally the last step of the spectral clustering algorithm, k-means, is parallelized. Proposed method by Zheng et al. achieved around ten times speedup using CUDA and one of the pioneer studies of the field.

Another GPGPU utilizing parallel spectral clustering algorithm is conducted by Ito et al. [6]. In their study a naïve boosting is performed by accelerating the computational bottleneck of the first step of the spectral clustering which is finding the k-nearest neighbors for all fragments and introduce a minor speed-up.

Beyond these mentioned studies there is no significant study that employs GPGPU on spectral clustering. In this study we analyze spectral clustering subroutines on GPGPU, with various utilizations in a stepwise manner. For each step, more than one approach to the utilization of GPGPU is provided also with a comparison of naïve host implementation. Stepwise analysis may reveal some attack points to the problem of graph partitioning on GPGPU and concentrate further studies to these sub-problems.

The remainder of this paper is organized as follows. In section III, spectral clustering algorithm on GPGPU is explained and analyzed in a stepwise manner. In section IV, we compared the performance of each utilized parallelization routine. Finally our techniques are concluded and discussed in section V.

III. Spectral Clustering on GPGPU

In this section spectral clustering algorithm is explained along with our approach of parallelization by GPGPU. Spectral Clustering can be conducted as six consecutive steps. Each step can be handled separately to achieve stepwise parallelization and can be considered as a separate module as shown in algorithm flow in figure 1.

A. Distance Matrix Generation (Step 1)

The first step is generating the neighbor graph as spectral-clustering is a graph theoretic clustering algorithm. As the dataset gets large and dimension of the data is high, finding the neighborhoods in the high-dimensional space requires a long time for the computation so the first step is vital to be parallelized to boost performance of the overall algorithm.

To overcome complexity it this step is implemented as to use brute force t- nearest neighbor search method accelerated with GPGPU. This step assumes that the data has already been read from the file and resides in the host memory also number of nearest neighbors t is given as a user defined input. Given n-by-d data matrix, where n is the number of data d is the number of dimensions; by considering number of nearest neighbors t, compute nxn distance S matrix by blocking the data. Meaning that first select a block of data, fetching in row order as to be fast, then compute Euclidean distance between block and data, and find nearest neighbors. As each block will be handled in parallel on GPU brute force t-nn will be boosted.

This step is utilized parallelization with GPGPU by considering the distance calculation problem as a matrix multiplication. This approach eliminates the problem of the feature dimension by calculating distances in one step in parallel. Since the final goal of this step is to find distance from every point to every other point regardless of the dimensionality, we have to calculate our distance metric for each point to every other.

The Euclidean distance between two points ***p*** and ***q*** is given by:

$$\|q - p\| = \sqrt{\|p\|^2 + \|q\|^2 - 2pq}. \tag{1}$$

Where $\| . \|$ corresponds to length of the vector, in order to calculate distance between each points as in eq. 1, it is needed to multiply feature matrix with its transpose to obtain a matrix F where sub-diagonals consist of ***pq***, super-diagonals consist of ***qp*** and as the diagonal ***pp*** or ***qq***. This matrix-matrix multiplication is conducted by the use of CUBLAS library [7] subroutine *cublasSgemm*. After this multiplication it is only needed to process matrix F with a kernel in one pass to calculate distance.

After obtaining the distance matrix the challenging part of this step arises this is the selection of *t*-nearest neighbor of each point in the distance matrix. Note that spectral clustering can be performed without selecting *t*-nearest neighbor of a point and dropping other points from the distance matrix but then the problem of dense distance matrix problem should be handled in further steps. This step makes the distance matrix sparse and decreases the complexity of the overall algorithm.

On the GPGPU perspective two different schemas used to utilize parallelization with GPGPU. Our goal is to find *t*-closest element of each row to the element on the diagonal of that row, and repeat for each row of the entire distance matrix. First approach utilizes CUDA-Thrust library [8] to sort each row of the distance matrix sequentially and further dropping except first t-elements of each row. CUDA-Thrust library is very fast and well-optimized for parallel operations on arrays and array typed data-structures but the lack of the block processing capability of matrices directly, lead us to use it sequentially. Of course sorting whole matrix as an array and processing the index intervals for rows seems as a solution, it cannot be completed even the number of elements in the distance matrix is 400x400 because of the memory bottlenecks Thrust library uses. The performance of this approach is illustrated in the table 1. It is expected to have a monotonically increasing time complexity as the number of rows increase. This drawback can be seen in the figure 2.

Second approach for parallelization of this step on GPGPU is to use a sorting routine for each row of the distance matrix on parallel. Insertion sort applied to each row of the distance matrix on parallel. This kernel is not optimized with shared memory or texture memory usage but noted as a further improvement. The details of this subroutine can be found in the study [9], our implementation uses the kernel mentioned in the study with slight modifications. As expected utilizing a naïve sorting algorithm in parallel improves the performance, also the increase of time complexity is sub-linear as the number of elements grown, this characteristic is shown in graph 1.

We can come to a conclusion that it is suitable to use CUDA-Thrust library when we are dealing with arrays or when we can transform our problem in array domain. It is observed by our experiments that using CUDA-Thrust library routines are even faster for perfect-parallelism used (ex. CUDA-Thrust performs faster than a direct kernel while filling elements of an array). When the problem domain cannot be transformed, it is more convenient to use fully parallelized naïve approaches than very fast sequential routines.

B. *Compute Affinity (Similarity) Matrix (Step 2)*

Similarity conversion of distance matrix is one of the major contributions of spectral clustering algorithm and many diversifications exist in the literature questioning which similarity metric should be used. In our implementation we employed most common kernel which is radial basis kernel in order to calculate similarities between data points.

This step is composed of computing the affinity matrix given a set of points $S = \{s_1 , . . . , s_n \}$ in $\mathbb{R}$ as converting distance matrix to a sparse similarity matrix which can also be called as affinity matrix. Forming the affinity matrix $A \in \mathbb{R}^{nxn}$ defined as follows:

$$\begin{aligned} A_{ij} &= e^{-\|s_i - s_j\|/2\sigma^2} && i \neq j \\ A_{ij} &= 0 && i = j \end{aligned} \tag{2}$$

This exponential is calculated for each point by considering the entries on distance matrix S and well suited for parallelization. The scaling parameter σ is some measure of when two points are considered similar. The selection of σ is commonly done manually but when the data contains multiple scales, even using the optimal σ fails to provide good clustering. Hence self-tuning version of the spectral clustering algorithm is implemented in this study. In self-tuning version, the selection of the local scale σ_i can be done by studying the local statistics of the neighborhood of point s_i. The selection is done by computing the distance between the point s_i with the mean of its row. Note that by not having any assumptions on the adjacency matrix all points are thought to be fully connected (e.g. *t*-nearest neighbor selection step skipped) also the scaling parameter σ^2 can be replaced by σ_i and σ_j if self-tuning spectral clustering as implemented in this study.

Self-tuning version needs to calculate mean of each row of the sparsified distance matrix. Dealing the mean operation sequentially may cause problems as stated in the previous step, thus we approached to the problem as a matrix-vector multiplication again. Calculation of mean needs to compute non-zero elements and sum of these elements for each row then dividing the result to further be used in calculation of local scaling parameter σ_i. The sum of elements of each row of a matrix can be computed by multiplying the matrix with a vector consists of ones. This generates a vector having row sums of matrix on its corresponding elements. Trivially by binarizing the input matrix, we obtain number of nonzero elements in each row and calculation of mean becomes easy to compute with two matrix-vector multiplications.

In this step a copy of distance matrix is binarized to compute non-zero elements before-hand by a kernel and matrix-vector multiplications are conducted by using CUBLAS routine *cublasSgemv*. After calculating means of each row it is straightforward to compute local scaling parameter as explained in the previous sub-section. This matrix-vector multiplications parallelizes the step perfectly and improvements can be seen by considering tables 1-3.

C. *Calculate Graph Laplacian (Step 3)*

Computing diagonal matrix D does not need intensive computing since matrix D consists of the sum of elements in a row of the affinity matrix A on its diagonals. It represents volume (degree) of a node in the overall adjacency graph to be partitioned.

Define D to be a diagonal matrix with $D_{ii} = \sum_{j=1}^{n} A_{ij}$ whose (i,i)- element is the sum of A's i-th row. This step is also very similar to calculating the mean of rows of a matrix and handled in that manner. A matrix-vector multiplication is conducted to compute row-sums by CUBLAS routine *cublasSgemv* and it is used to build the diagonal matrix D which will be multiplied on left and right of the similarity matrix S by CUBLAS routine *cublasSgemm.* The performance of this step brings minor improvements compared with the cpu implementation as illustrated in tables 1-3.

D. *Calculate Eigen values and Eigen vectors of Laplacian Matrix (External Step)*

Efficient and parallel Eigen value decomposition is one of the major topics of scientific, parallel computing community and many algorithms exist to find k-leading Eigen values and Eigen vectors of a square-symmetric matrix. On the side of GPGPU community there does not exist many implementations of *evd* moreover the existing ones have preconditioned states. The implementation in CUDA-ZONE has the precondition of tri-banded-matrices and another study [10] has some platform dependent requirements. It is out-of scope of this study to analyze GPGPU on *evd* problems but noted as a future study. Because of the mentioned points *evd* routines are conducted on host-cpu side and results are feed to GPGPU steps.

E. *Calculate Graph Laplacian (Step 4)*

By the assumption of having Eigen values and Eigen vectors of the Laplacian matrix our next task is to select k-largest Eigen values and their corresponding Eigen vectors with assumption that host-cpu implementation provides all Eigen values and Eigen vectors of the Laplacian matrix. Calculation of k-largest Eigen values is conducted by using CUDA-Thrust library and by using sorted Eigen values k-columned matrix X is generated by a separate kernel. This step is a minor step and can be considered as a preprocessing step of K-means step. Generated matrix X is normalized by several steps on GPGPU. Renormalization of matrix L is essential for further k-means clustering step. Renormalize the rows of X to have unit length yielding $Y \in \mathbb{R}^{nxk}$, such that $Y_{ij} = X_{ij} / (\sum_j X_{ij}^2)^{1/2}$. These steps are conducted by first multiplying the matrix X with its transpose to further extract its diagonal elements as to form normalization vector. In this step for the matrix multiplication CUBLAS routine *cublasSgemm* is used.

F. *K-means Clustering of the rows of Y (Step 5)*

K-means step is the final step of the spectral clustering step. As the last step any clustering algorithm might be used but the simplicity of k-means makes it the most common used clustering in spectral-graph partitioning closer. In this step k-means algorithm used with two different utilization schemas one with utilization of shared memory reduction with texture and the other one is block shared memory usage. Both have their advantages and drawbacks, as the number of clusters is low, texture memory utilizing k-means suffers from low shared memory usage, and by the opposite, as the number of clusters increase block shared memory utilized version initiates consecutive loops to fill shared memory. As the former k-means we used our own implementation and for the latter version the implementation in [11] is used. The performance analysis of K-means step is illustrated in tables 1-3 and also analyzed in the next section.

IV. Experiments and Performance Analysis

Experiments are conducted in three different configurations namely, two different optimization techniques on GPGPU side and a baseline implementation on host-cpu side. Details of the steps are as follows:

Configuration 1:

1) Step 1.1 thrust::sort_by_key
   Step 1.2 memory operations (host to device)
   Step 1.3 kernel executions
   Step 1.4 memory operations (device to host)
   Step 1.5 matrix symmetrization
   Step 1.6 setting nearest neighbors on distance matrix
   Step 1.7 thrust::sequence
2) Step 2.1 memory operations (host to device)
   Step 2.2 kernel executions
   Step 2.3 memory operations (device to host)
3) Step 3.1 kernel executions
   Step 3.2 memory operations (host to device)
   Step 3.3 memory operations (device to host)
4) Step 4.1 conversion of eigenvector matrix
   Step 4.2 thrust::sort
   Step 4.3 thrust::sequence
5) Step 5.1 normalize matrix rows
   Step 5.2 cuda-kmeans using block shared mem. opt.

The second configuration is as follows:

1) Step 1.1 cuda insertion sort
   Step 1.2 matrix symmetrization
   Step 1.3 setting nearest neighbors on distance matrix
   Step 1.4 kernel executions
   Step 1.5 memory operations (host to device)
   Step 1.6 memory operations (device to host)
2) Step 2.1 kernel executions
   Step 2.2 memory operations (host to device)
   Step 2.3 memory operations (device to host)
3) Step 3.1 kernel executions
   Step 3.2 memory operations (host to device)
   Step 3.3 memory operations (device to host)
4) Step 4.1 conversion of eigenvector matrix
   Step 4.2 thrust::sort
   Step 4.3 thrust::sequence
5) Step 5.1 normalize matrix rows
   Step 5.2 cuda-kmeans using texture opt.

The third configuration is a naïve host-cpu implementation with no optimizations. The eigen decomposition step is excluded as step 4. In the host-cpu side implementation following study is used [12] proposed by Chen et.al. Configuration 3 is given as only with major steps:

1) Step 1 generate t-nearest distance matrix
2) Step 2 generate similarity matrix
3) Step 3 generate graph laplacian
4) Step 5 host-k-means

The analysis of the GPGPU is considered first by comparison of two different configurations on GPGPU, starting with the first steps of the configuration 1 and 2. The comparison of the two different sorting algorithms are illustrated in the figure 2 and it can be seen that, as the number of data samples increase, insertion sort implemented on GPGPU by parallel meets the thrust sort routine. As a linearly increasing function of thrust sort can be used if there is a lack of optimized parallel sorting algorithm of matrix rows.

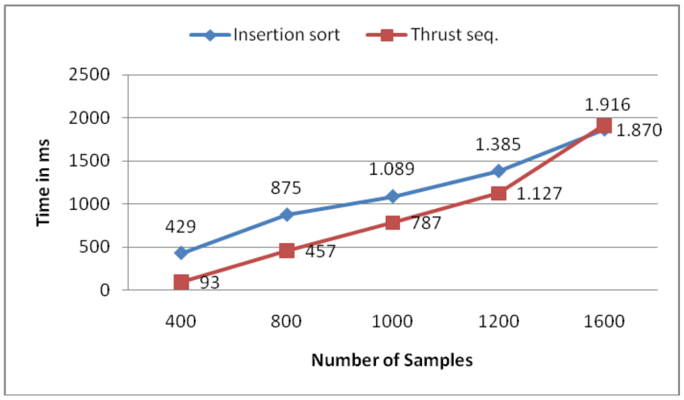

Figure 2 . CUDA Thrust sort vs Insertion sort

Also it can be seen that host-cpu implementation is far from comparison with GPGPU implementations as illustrated in figure3.

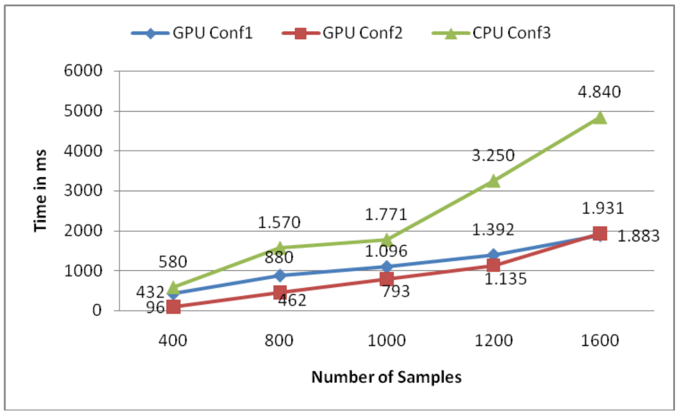

Figure 3 . Total time of all three configurations in step1

The second and third steps of the spectral clustering algorithms has no divergence between configurations hence the overall stepwise comparison is given only as shown in figure 4-5 below.

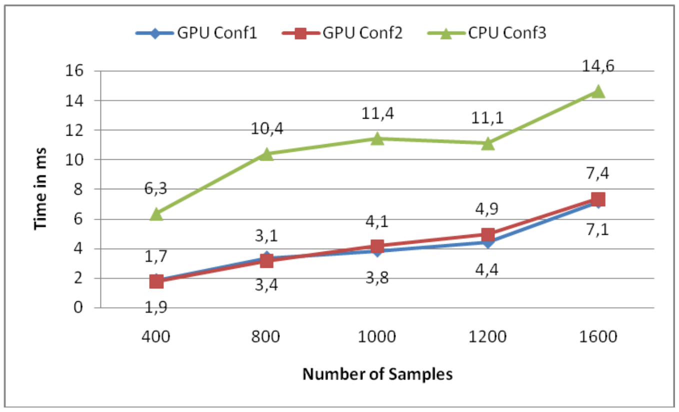

Figure 4 Total time of all three configurations in step2

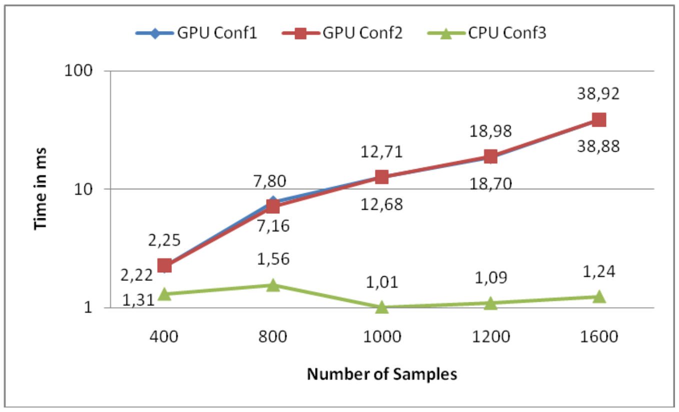

Figure 5. Total time of all three configurations in step3

Figure 5 points out an interesting characteristic of step 3 namely, *Graph Laplacian Calculation* step which consists of matrix-vector and matrix-matrix multiplications. In this step as stated in the previous section all linear algebra routines are executed by CUBLAS library routines but the host-cpu implementation outperforms the GPGPU implementation of this step expresses that it is much more reasonable to use host-cpu implementation in this particular step.

Step 4 is consists of Eigen vector concatenations to form Eigen vector matrices to be used in K-means algorithm and does not have any difference between two configurations. This step is excluded from performance analysis of host-cpu implementation; further details can be seen in table 1-3.

Step 5 executes k-means clustering on the matrix composed of Eigen vectors. As stated in the previous section two different k-means implementation employed in this step along with host-cpu implementation. The corresponding sub step is step5.2 of configuration 1 and 2 as also illustrated in figure 6 below.

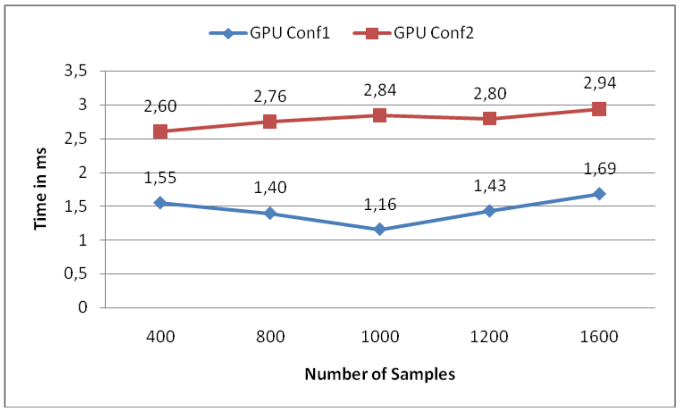

Figure 6 Block shared memory vs Texture utilized k-means

Figure 6 illustrates that using shared memory blocks is faster that using texture memory in some extent. It should be considered that as the number of samples increase, block shared memory version will suffer from memory transfers as it will be constant for utilized texture memory. Another structure to point out is that the texture version of k-means is much more stable than the other, the overall k-means step can be seen in figure 7 including the host-cpu implementation as well.

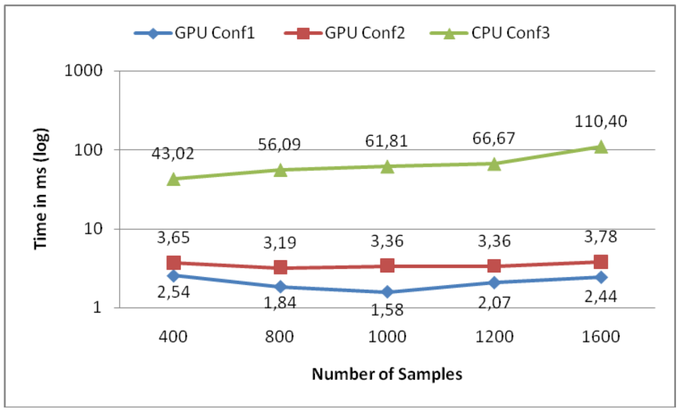

Figure 7 Total time of all three configurations in step5

The overall running time of all three configurations is illustrated in figure 8 excluding *evd* step from all configurations.

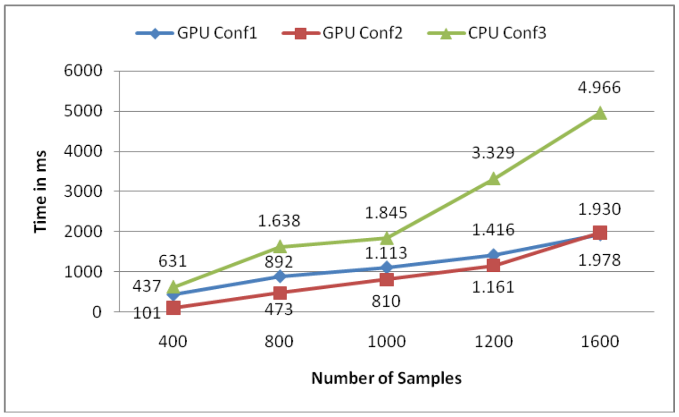

Figure 8 Total overall time of all three configurations

All the experiments are conducted on a Intel i7 CPU 870 @2.93Ghz with 4GB memory, and a Tesla C2070 NVIDIA GPU installed with CUDA Runtime Version 4.0 and compute capability 2.0. Number of threads and block sizes change respectively only for sequential operations. Except Step 1.1-"cuda insertion sort", Step 5.1-"normalize matrix rows" and some internal kernel calls in Step 5.2 "cuda-kmeans"; number of threads per block fixed at 256 and block size varies with respect to number of samples.

V. Discussion & Conclusion

In this study we analyzed spectral graph partitioning sub-routines with various approaches on GPGPU and analyzed their performance with a further implementation on host-cpu. The major assumption of spectral clustering is well suited for GPGPU implementation is met as analyzed in the previous section. That is without some minor steps all other steps outperform naïve host-cpu implementation. The details of the spectral clustering are analyzed step by step and the most significant step found out to be the step 1 of each differing configurations. Calculation of t-nearest neighbors in a large dense distance matrix is the most challenging part of spectral clustering which will be attacked in out further studies. Another point is that implementing *evd* and analyzing its affects to the overall algorithm must be handled on a GPGPU perspective which is also noted as a future study.

CUBLAS library is extensively used in this study and found very useful in matrix routines which are the main axis of spectral clustering. CUBLAS library routines are used only in dense matrices which should be implemented as a sparse version and by the use of CUSPARSE library, the number of data points should be increased to be able to use GPGPU implementation extensively. Most of the matrices generated in this study has sparse variant that can be utilized and boost performance.

CUDA-Thrust library is also a very useful tool when our problem consists of array-typed data structures. Developing matrix-block operations by using CUDA-Thrust is expected to boost performance also and will be further analyzed.

APPENDIX

All of the performance results are illustrated below.

Num samples	Step1								Step2				Step3				Step4				Step5			TOTAL
	Step1.1	Step1.2	Step1.2	Step1.3	Step1.4	Step1.5	Step1.6	Total	Step2.1	Step2.2	Step2.3	Total	Step3.1	Step3.2	Step3.3	Total	Step4.1	Step4.2	Step4.3	Total	Step5.1	Step5.2	Total	
400	428,7142	1,054528	1,43856	1,21584	0,02816	0,007808	0,010496	432,4696	0,37568	1,068064	0,406848	1,850592	1,068064	0,748288	0,406848	2,2232	0,006816	0,4464	0,0056	0,458816	0,989184	1,552	2,541	437,0022
800	875	2,989376	0,740096	1,218816	0,091584	0,016768	0,031072	879,7748	1,23632	1,495008	0,64624	3,377568	5,305056	1,25472	1,237888	7,797664	0,007616	0,594528	0,0056	0,607744	0,438048	1,397056	1,835	891,5578
1000	1089,349	3,644704	0,99136	1,56416	0,155808	0,02	0,046784	1095,772	1,641536	1,619328	0,570976	3,83184	9,849216	1,2472	1,583808	12,68022	0,006976	0,322048	0,005632	0,334656	0,419616	1,157056	1,577	1112,619
1200	1384,73	5,007968	1,03008	1,290656	0,199392	0,025504	0,065344	1392,349	2,407968	1,237056	0,773376	4,4184	14,72045	1,424224	2,558176	18,70285	0,007584	0,857536	0,005632	0,870752	0,641696	1,430048	2,072	1416,341
1600	1870,251	9,43472	1,416736	1,652288	0,36528	0,034688	0,114016	1883,269	4,42784	1,67808	1,029056	7,134976	32,63689	1,79392	4,44848	38,87929	0,008608	1,053728	0,005696	1,068032	0,751424	1,685696	2,437	1930,351

Table 1. Running time analysis of overall configuration 1

Num samples	Step1							Step2				Step3				Step4				Step5			TOTAL
	Step1.1	Step1.2	Step1.3	Step1.4	Step1.5	Step1.6	Total	Step2.1	Step2.2	Step2.3	Total	Step3.1	Step3.2	Step3.3	Total	Step4.1	Step4.2	Step4.3	Total	Step5.1	Step5.2	Total	
400	92,9593	0.027936	0.007136	1,189056	0,789856	1,34976	96,28797	0,985024	0,40496	0,34464	1,734624	1,271168	0,636928	0,339744	2,24784	0,007072	0,456	0,005504	0,468576	1,043584	2,602848	3,646432	100,7
800	457,0616	0,093184	0,018368	0,671104	0,847584	3,60736	462,2992	1,366304	0,61536	1,159488	3,141152	5,15616	0,8832	1,1184	7,15776	0,007552	0,638432	0,005664	0,651648	0,43488	2,755468	3,190348	473,2
1000	786,7182	0,15424	0,023072	0,76368	0,917632	4,25696	792,8338	1,765632	0,624736	1,747904	4,138272	9,826976	1,168064	1,711904	12,70694	0,006944	0,532896	0,005632	0,545472	0,516384	2,844672	3,361056	810,2
1200	1127,015	0,203232	0,027648	0,991776	1,319392	5,912704	1135,469	1,374592	0,947264	2,62416	4,946016	14,76598	1,659424	2,555392	18,9808	0,00768	1,158016	0,005696	1,171392	0,56384	2,795412	3,359252	1161
1600	1916,386	0,361952	0,036928	1,531488	1,470304	11,09424	1930,881	1,80368	1,012576	4,534688	7,350944	32,65139	1,72176	4,543072	38,91623	0,008608	1,108064	0,005664	1,122336	0,840096	2,941196	3,781292	1978

Table 2. Running time analysis of overall configuration 1

Num samples	Step1	Step2	Step3	Step4	Step5	Total
400	580	6	1,30697	0	43,02036	631
800	1570,196	10,37924	1,561738	0	56	1.638
1000	1770,821	11	1,00974	0	61,81272	1.845
1200	3250,461	11,09968	1	0	67	3.329
1600	4840,15	14,63657	1,243511	0	110,3958	4.966

Table 1. Running time analysis of overall configuration 3

ACKNOWLEDGMENT

The authors would like to thank the METU Informatics Institute Game Technology Department staff and Virtual Reality and Computer Vision Research Group for making it possible to use CUDA enabled PCs of Vision-Lab.

REFERENCES

[1] Ng, A. Y., Jordan, M. I., and Weiss, Y., "On spectral clustering: Analysis and an algorithm," In Advances in Neural Information Processing Systems (NIPS),(2001)

[2] Shi, J. And Malik, J., "Normalized cuts and image segmentation", IEEE Trans. Pattern Anal. Mach. Intell. (2000)

[3] Jing Zheng, Wenguang Chen, Yurong Chen, Yimin Zhang, Ying Zhao, and Weimin Zheng. "Parallelization of spectral clustering algorithm on multi-core processors and GPGPU". In 2008 13th Asia-Pacific Computer Systems Architecture Conference (ACSAC), page 8, Piscataway, NJ, USA, August 2008. Tsinghua Univ., Beijing, China, IEEE.(2008)

[4] Miao G,Song Y,Zhang D,Bai H ."Parallel spectral clustering algorithm for large-scale community". *In: The 17th workshop on social web search and mining (SWSM)*(2008)

[5] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, Edward Y. Chang, "Parallel Spectral Clustering in Distributed Systems," *IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 568-586, March*, (2011)

[6] Ito, J., Junichi Higo, Tomii, K.: "Classification of Protein Fragments with a Spectral Graph Technique", *The 20th International Conference on Genome Informatics, pp.P034-1-P034-2*(2009).

[7] http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUBLAS_Library.pdf

[8] http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/Thrust_Quick_Start_Guide.pdf

[9] V. Garcia and E. Debreuve and M. Barlaud. Fast k nearest neighbor search using GPU. *In Proceedings of the CVPR Workshop on Computer Vision on GPU, Anchorage, Alaska, USA, June 2008*

[10] Bryan Catanzaro, Bor-Yiing Su, Narayanan Sundaram, Yunsup Lee, Mark Murphy and Kurt Keutzer, "Efficient, High Quality Image Contour Detector", International Conference on Computer Vision (ICCV), September 2009

[11] http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html

[12] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y. Chang . "Parallel Spectral Clustering in Distributed Systems" EEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), Vol. 33, No. 3, pp. 568-586, March 2011