# Parallelization of Spectral Clustering Algorithm on Multi-core Processors and GPGPU*


Jing Zheng
*Tsinghua University, Beijing, China*
zheng-j06@mails.tsinghua.edu.cn

Wenguang Chen
*Tsinghua Uniersity, Beijing, China*
cwg@mail.tsinghua.edu.cn

Yurong Chen
*Intel China Research Center, Beijing, China*
yurong.chen@intel.com

Yimin Zhang
*Intel China Research Center, Beijing, China*
yimin.zhang@intel.com

Ying Zhao
Tsinghua University, Beijing, China
yingz@mail.tsinghua.edu.cn

Weimin Zheng
*Tsinghua University Beijing, China*
zwm-dcs@mail.tsinghua.edu.cn



## Abstract

*Spectral clustering is a widely-used algorithm in the field of information retrieval, data mining, machine learning and many others. It can help to cluster a large number of data into several categories without requiring any additional information about the dataset or the categories, so that people can find information by categories easily. In this paper, we parallelize the algorithm proposed by Andrew Y. Ng, Michael I. Jordan and Yair Weiss. We provide two versions of implementation: one is parallelized in OpenMP; the other is programmed in the NVIDIA CUDA (Compute Unified Device Architecture), which is the environment provided by NVIDIA to program on its CUDA-Enabled GPGPUs (General-Purpose Graphic Processing Unit). We can achieve about three times speedup in OpenMP and around ten times speedup using CUDA in our experiments.*


## 1. Introduction

Spectral clustering was first proposed by Donath and Huffman to study graphic partitions in 1973. In the following decades, many researchers contributed to promote its performance and extended it to many other fields. Thanks to the great work of Shi and Malik(2000), and Andrew Y. Ng et al. (2002), spectral clustering became well-known in information retrieval fields[1].

Spectral clustering is very useful in information retrieval. For instance, as a result of the development of Internet, there are so many data on it that information needed is sometimes difficult to find. Figure 1 shows the search results responding to "bacon" by Google Search Engine. There are about 39,600,000 results found. It is impossible to look through all these items. However, if we analyze the first few items, we can find that the results can be divided into two categories: one is about the famous person named Francis Bacon; and the other is about a kind of food. If these two categories of items can be separated, it would help users to find the information they need more conveniently. Spectral clustering algorithm can cluster these items into two clusters according to the content of them. The algorithm implemented in this paper aims at document clustering. However, as an effective clustering algorithm, spectral clustering can be used widely in image or multi-media search and many other fields, as well.

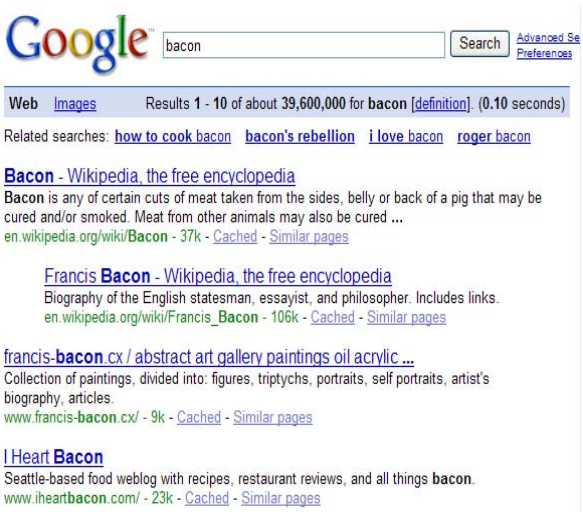


**Figure 1. Google search results for bacon**


978-1-4244-2683-6/08/$25.00 ©2008 IEEE

Spectral clustering outperforms than other clustering algorithm such as *kmeans* at the cost of higher complexity. At least $O(k*n^2)$ ($k$ is the number of clusters, while $n$ is the scale of dataset) multiplications are needed to compute eigenvectors of a dense symmetric matrix. As a result, when n is over 3000, it takes about a minute to finish clustering. It is too long for a real time system such as a search engine. The parallelization of the algorithm can reduce the responding time to make it more acceptable.

In addition, as GPGPUs do not support double precision floating point operations, we check all the test datasets we could obtain and the clustering in single precision achieves the same result as in double precision. It is probably because the matrix consisted of the first few eigenvectors is an approximation to the spectral matrix of the input matrix. As spectral clustering is not precision-sensitive, we can implement it on GPGPUs without any accuracy loss.

During recent years, GPGPU has been developing rapidly. Since NVIDIA released CUDA in February 2007, it has attracted the attentions of many researchers in all kinds of communities such as weather prediction [4], molecular dynamics [5], fluid dynamics [6], and so on. Implementing spectral clustering on CUDA can help us to learn more about the performance of CUDA in the field of data mining.

The purpose of our work is to parallelize spectral clustering algorithm and evaluate its performance. We implemented it in two parallel programming models: one is in OpenMP, and the other is in CUDA. We test the two implementations and obtain acceptable speedup.

The rest of this paper is organized as follows: Section 2 introduces related works. Section 3 explains the algorithm, a simple analysis to help parallelization and the parallelization of the algorithm. In section 4 we describe the implementation in OpenMP and in CUDA. Experiment results are presented in section 5. Finally, conclusions and Future work are discussed in Section 6.

## 2. Related work

### 2.1 Spectral clustering algorithm and its parallelization

Since the 1990s, spectral clustering has been extended to more fields from graph partition. Thanks to the excellent works of Shi and Malik (2000) and Andrew Y. Ng et al. (2002), spectral clustering became well known in information retrieval domain because of its better performance and acceptable complexity.

According to the Laplacian matrix used in the algorithm, spectral clustering is divided into two categories: unnormalized spectral clustering and normalized spectral clustering. The main difference between these two kinds of algorithms is whether the matrix L normalized [1]. The algorithm that we parallelize is a normalized method.

Few works about parallelization of clustering are found. However, eigenvalue problem of symmetric matrix, which is an important part of the algorithm, is studied by many researchers. Some efficient methods are discussed in [7, 8, 9, 10, 21]. The most popular algorithm to compute eigenvalues and the respective eigenvectors of a symmetric matrix is IRLM (Implicitly Restarted Lanczos Method) [8]. IRLM is used in most high performance linear libraries such as LAPACK, MATLAB and ARPACK. However, most of these libraries are sequential. The latest math library parallelized in OpenMP is Intel MKL (Math Kernel Library) 10.0 [20]. MKL implements most functions of BLAS (Basic Linear Algebra Subprograms) and LAPACK, which are both efficient math libraries and contains useful basic function in scientific computing. We compared our implementation with MKL in section 5.

### 2.2 GPGPU and CUDA

NVIDIA G80 series are designed to meet the demand of programmable GPGPUs. It contains hundreds of cores and broad memory bandwidth. Moreover, unlike other GPGPUs, programming on NVIDIA G80 series does not require special programming language or execution through graphics APIs with support of CUDA [12].

CUDA environment is created to develop on the NVIDIA GPUs. It supports interfaces in C language and Fortran language, both of which are very popular in high performance computing.

Last year, many researches on CUDA showed its potential in various fields of general purpose applications such as fluid dynamics, molecular dynamics, as well as bioinformatics [13].

## 3. Spectral clustering and its parallelization

The spectral clustering algorithm in this paper was proposed by Andrew Y. Ng, Michael I. Jordan and Yair Weiss in 2002[14]. The detail of the algorithm is as follows:

*Input: document-word matrix $S \in R^{n \times m}$, number k of clusters to construct, and scaling parameter $\sigma$*

1. *Construct an affinity matrix A.*
2. *Compute diagonal matrix D, whose i-th element*

*in diagonal is the sum of the i-th row of matrix A*

3. *Compute Laplacian Matrix* $L=D^{-1/2}AD^{-1/2}$.
4. *Compute the largest k eigenvectors* $v_1, v_2, \ldots, v_k$ *of L, and let V be the matrix composed with* $v_1, v_2, \ldots, v_k$ *as columns.*
5. *Normalize matrix V by rows. Let X to be the normalized matrix.* $x_{ij}=x_{ij}/(\sum x^2_{ij})^{1/2}$
6. *Clustering the rows of X into k clusters, regarding each row of V as a point in k-dimension space.*

In [14], matrix A is computed according to formula 1. $||s_i-s_j||$ $(i \neq j)$ can be computed by multiplying the *i*-th and *j*-th row of the sparse input matrix S. And the first step of spectral clustering can be considered as a series of sparse vector-vector multiplications. Consequently, this algorithm mainly consists of matrix and vector operations. Matrix problems usually imply huge possibility to parallelization because they are compute-intensive and matrices are able to be divided by rows, columns or blocks. So we can draw the conclusion that spectral clustering is well-suited to be parallelized.

$$A_{ij} = \begin{cases} e^{-\|s_i - s_j\|^2 / 2\sigma^2} & i \neq j \\ 0 & i = j \end{cases} . \qquad \textbf{Formula 1}$$

### 3.1 Analysis of spectral clustering

Figure 2 is the flow chart of the algorithm. To make the program more organized and easier to analyze, we divide the program to three steps: *getAffinityMatrix, spectralComputing,* and *kmeans*, based on the property of matrixes operated in them. The matrix processed is a sparse matrix in step *I*, a square symmetric matrix in step *II*, and a matrix whose second dimension is reduced to *k* in step *III*.

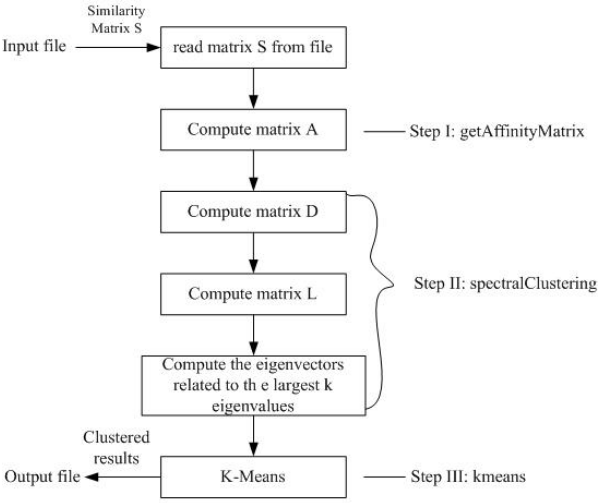


**Figure 2. Flow Chart of sequential program**

Figure 3 shows the percentage of time that each function occupies. The datasets *tr23, k1b,* and *sports* are three of CLUTO [19] standard test datasets. *Tr23* contains 204 documents; *k1b* contains 2340; and *sports* contains 8580. It is obvious that the first two steps take over 90% on the three datasets of different sizes. Moreover, *spectralComputing* occupies most execution time when the number of documents is large enough. As a result, during parallelization, we focus on the first two steps, especially the second one. Assuming the first dimension of matrix S is n, the complexity of step II is about *O(n*n*k)* to compute the first *k* eigenvectors.

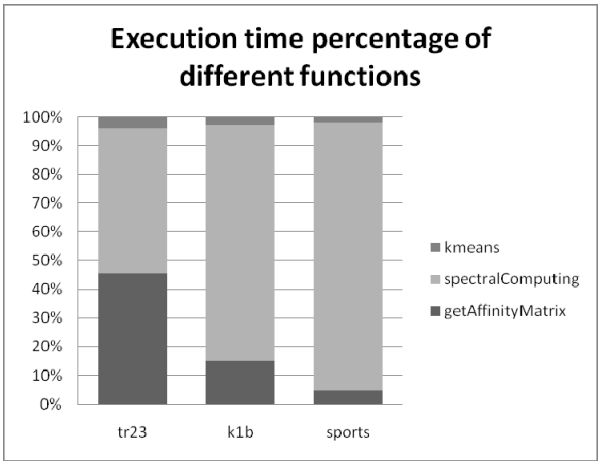


**Figure 3. time of different funtions**

### 3.2 Parallelization of spectral clustering

Now we elaborate each step described in section 3.1. In step *I*, the n×n symmetric matrix A is computed by multiplying every two rows of sparse matrix S with n rows and m columns. The procedure can be described

as follows:

```
for i=0, 1,…, n do
   for j=i, 1, …, m do
      if i=j
        a_ij ← 0;
      else
        a_ij ← s_i * s_j
```

There are two loops in step I either of which be parallelized. To achieve better spatial locality and lower cache miss, the inner loop is parallelized. To achieve load balance, we group rows according to the number of non-zero elements they contain.

Step *II*, *spectralComputing*, computes Laplacian Matrix *L* and its eigenvectors corresponded to the first *k* largest eigenvalues of *L*. As a symmetric real matrix, *L*'s eigenvalues and respective eigenvectors can be computed by IRLM, which is widely used in most high performance linear libraries such as ARPACK, PROPACK and MATLAB. However, these libraries require users to provide the function that implements dense matrix-vector multiplication. Our efforts on *spectralComputing* focus on the parallelization of dense matrix-vector multiplication.

Step *III* is the classical clustering method which is described as follows. Although it takes much less time than the step *I* and step *II*, we also parallelize it and obtain speedup that is approximately linear. The parallelization of the loop in *2* in the following algorithm means that each thread computes the cluster number of one point.

*K-Means Algorithm*

*Input: n points in m-dimension space, number of clusters*

*1. Generate k random points as the initial center of the k clusters*

*2. For i=0, 1, …, n, compute the distance between the i-th point and each cluster center and put the point into the cluster that the distance is shortest.*

*3. Compute the new centers of the k clusters*

*4. If the new centers are the same as the old ones, Stop; if no, take the new ones as centers of clusters and then go to 2*

# 4. Implementation

## 4.1 Data structure

Sparse matrix *S* is stored in CSR(Compress Sparse Row) format. We use three arrays to represent it:

*val* --a single array contains all the non-zero elements of sparse matrix *S*.

*idx* -- an integer array contains the column index of each non-zero elements.

*row* – an integer array contains the location of the first element in every row in array *val*.

The other matrixes in this program are dense and symmetric except the diagonal matrix *D*. These matrixes are all stored in one-dimension array in package format. In CUDA BLAS, matrix should be stored major in column. Fortunately, matrix *A, D* and *V* are all symmetric matrices, so the transposition is not necessary in our program.

## 4.2 Implementation in OpenMP

As analyzed in section 3, the spectral clustering algorithm can be easily parallelized with OpenMP compiler directives such as "parallel for". To reach higher performance, we also use BLAS in Intel MKL which is thread-safe.

## 4.3 CUDA programming model

In the NVIDIA CUDA programming model [15], the system contains a host which is usually a CPU and at least one GPU which is a highly-parallel coprocessor [16]. The host is responsible to control and memory management, while GPU creates thousands of threads to finish the work assigned by the host.

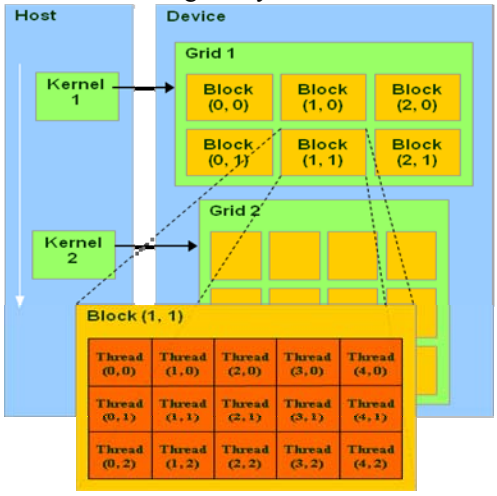


**Figure 4 [16].** CUDA programming Model

Threads on GPU are organized in the form of blocks, and blocks are grouped to Grids. Blocks can be 1-, 2- or 3-dimension; grids can be 1- or 2- dimension. This makes the partition of data more flexible and can be determined by developers according to data structure.

Figure 4 presents the programming model of CUDA intuitively.

**Table 1. Memory Properties of GeForce 8800GTS**

| | Location | Size | Hit Latency | Read-Only | Program Scope |
|---|---|---|---|---|---|
| Global | Off-chip | 640MB | 200~300 cycles | No | Global |
| Shared | On-chip | 16KB per SM | ≈ register latency | No | Function |
| Local | Off-chip | Up to 640MB | 200~300 cycles | No | Function |
| Constant | On-chip Cached | 64K | ≈ register latency | Yes | Global |
| Texture | On-chip Cached | Up to global | >100 cycles | Yes | Global |

NVIDIA Ge80 Series have a flexible but complicated memory model. The memory model of CUDA is composed of 5 layers: global memory, shared memory, local memory, constant memory, and texture memory. Developers can explicitly declare variables in the assigned memory. The layout of data in different memories might lead to varied performance. Shared memory is a new feature introduced, and it makes SMs work more efficiently. From Table 1, the hit latency of shared memory is near to registers. So developers should try their best to put frequently-visited data into shared memory and read-only data in texture or constant memory to minimize latency of memory access.

### 4.4 Implementation in the NVIDIA CUDA

Spectral clustering is a matrix-based application which is well-suited to CUDA. In addition, CUDA provides CUBLAS which is an implementation of BLAS. CUBLAS makes it easier to implement complicated matrix transformations and factorizations. CUBLAS is an implementation of BLAS on top of the NVIDIA CUDA driver. It allows access to the computational resources of NVIDIA GPUs [17].

When computing the affinity matrix $A$, we implement the CUDA function that computes a sparse matrix multiply itself in CSR format. We use hundreds of threads, and each of them computes a submatrix of matrix $A$. This method might be limited by the memory bandwidth as well as the conflicts between cache banks. We are trying to find better ways to solve this problem.

Compared to *getAffinityMatrix*, the second step *spectralClustering* takes much more time, when the number of documents is over 1000. We make our best efforts to port the function *spectralComputing* to CUDA. *SpectralComputing* computes the eigenvectors related to the first k largest eigenvalues of matrix L. As a symmetric real matrix, L's eigenvalues and related eigenvectors can be computed with IRLM[18]. IRLM, which is the used in MATLAB, is one of the most efficient algorithms to solve symmetric eigenvalue problems.

We ported the IRLM to CUDA based on APIs provided by CUBLAS. The performance of Lanczos depends greatly on the implement of matrix-vector multiply. In a sense, to parallelize IRLM depends greatly on parallelization of dense matrix-vector multiplication. We create hundreds of blocks, each of which computes one element of the result vector.

## 5. Experiments

To evaluate our parallelization, we conduct a series of experiments on several platforms. We evaluate the OpenMP version on the platform with four 4-core Genuine Intel 2.4GHz CPUs and 12GB main memory. The CUDA version is tested on two platforms: one is NVIDIA 8800GTS connected to a host with a four-core Genuine Intel 2.66GHz CPU and 4GB main memory, and the other is NVIDIA S870 connected to a host with two Dual-core AMD Opteron Processor 2216 and 8G main memory.

### 5.1 Test datasets

To evaluate the performance of the program, we use several CLUTO [19] datasets as inputs. The scale of these datasets (number of documents contained) is shown in Table 2.

**Table 2. datasets in the experiments**

| dataset | Number of documents | Sparse ratio |
|---|---|---|
| k1b | 2340 | 0.0068 |
| mm | 2521 | 0.0015 |
| la2 | 3075 | 0.0047 |

### 5.2 Experiments on OpenMP implementation

We evaluate our OpenMP implementation on a server with four 4-core CPUs. Figure 5 shows the speedup of the above datasets. When the number of cores is less than four, the speedup is approximately linear. However, as the number of cores increases, the speedup is limited by the memory bandwidth.

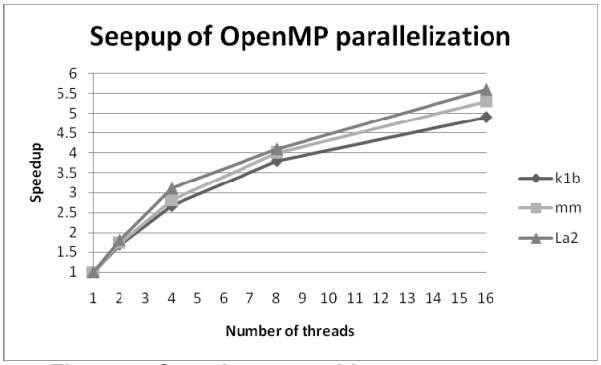


**Figure 5. Speedup on multi-core processors**

In addition, the speedup of step *I*, *II*, and *III* is tested separately. Figure 6 illustrates that *kmeans* has the best speedup among the three steps. *getAffinityMatrix,* the step which consists of sparse vector multiplications has the worst scalability among the three steps. That is probably because of the special properties of sparse matrices. The performance of *spectralComputing* step is between the other two steps. The data partition and element access of dense matrices is much more uncomplicated than that of sparse matrices. As a result, step *spectralComputing* has better speedup than step *getAffinityMatrix*. On the other hand, some codes of *spectralComputing* cannot be parallelized while most part of step *kmeans* can be parallelized. According to Amdahl Law, its speedup is limited. Consequently, the scalability and speedup of *spectralComputing* is worse than *kmeans*.

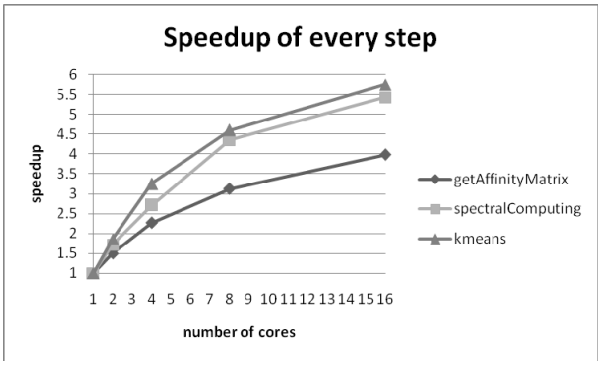


**Figure 6 Speedup of different functions**

## 5.3 Experiments on CUDA

We evaluate our CUDA implementation on GeForce 8800GTS and Tesla S870. The former is a graphic card supporting the CUDA environment and the later is a server containing four GPGPUs of Tesla architecture.

GeForce 8800GTS contains 12 eight-core 600MHz stream multiprocessors and 640MB global memory. The shared memory of each multiprocessor is 16KB which is divided in 16 banks. In the experiments, Geforce 8800GTS collaborates with an Intel Core2 Quad 2.4GHz sharing 4G memory. S870 is a 4-GPU server which support CUDA environment. Each GPU has 1.5GB global memory. S870 is connected to a host with two dual-core AMD Opteron Processor 2216 and 8G main memory.

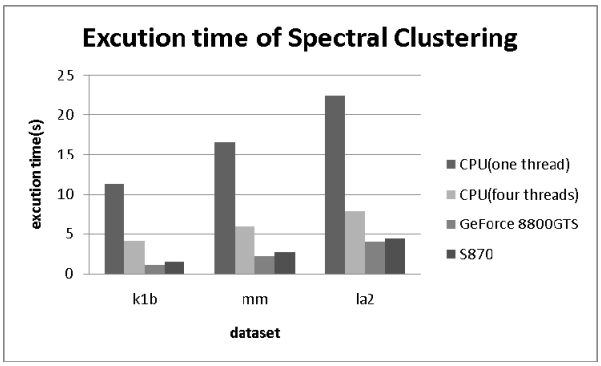


**Figure 7. Time of matrix-vector multiplication**

Figure 7 shows the performance of the matrix-vector multiply on a multi-core CPU and in the CUDA. In the former environment we use Intel MKL, while in CUDA we utilize CUBLAS library. It is obvious that CUDA BLAS library can improve the function two to three times than four threads on CPU and around ten times in the CUDA.

Thanks to the efficient CUBLAS, performance of spectral clustering on CUDA is enhanced greatly. Figure 8 shows the speedup on CPU and GPU. There is not as much improvement of performance as dense matrix-vector multiplication. It might result from other parts of the program like computing the affinity functions which contains a lot of sparse matrix operations. In addition, GeForce 8800GTS has 32 less cores and smaller memories than GeForce 8800GTX. As a result, the performance is not as effective as some released performance report of NVDIA CUDA 8800GTX.

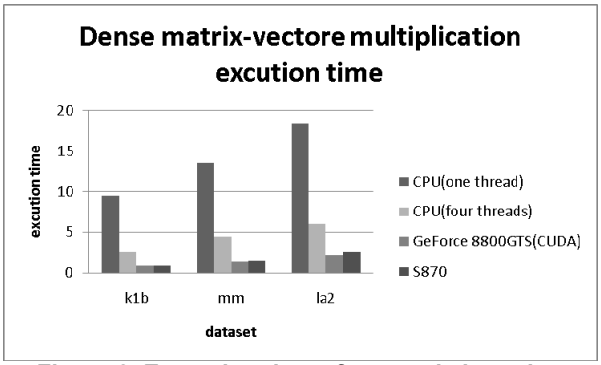


**Figure 8. Execution time of spectral clustering**

Compared to GeForce 8800GTS, the performance on S870 is a little worse. The reason is that data transmission between the main memory and the device memory costs more time on S870. Table 3 illustrates the bandwidth to transform matrix of different sizes on GeForce 8800GTS and Tesla S870. We tested the bandwidth when executing function CudaMemcpy() from host to device. It is obvious that 8800GTS achieves better bandwidth. As a result, when the matrix can be totally loaded into device memory, 8800GTS has better performance than S870.

**Table 2. Bandwidth from the main memory to the device memory**

| Matrix size | GeForce 8800GTS (GB/s) | Tesla S870 (GB/s) |
|---|---|---|
| 1000*1000 | 2.88 | 1.64 |
| 2000*2000 | 3.56 | 1.86 |
| 3000*3000 | 3.76 | 2.41 |
| 4000*4000 | 3.74 | 3.06 |

On the other hand, it is interesting that the data transmission from the device memory to the main memory is opposite to the transmission from the main memory to the device memory. The bandwidth from device memory to the main memory is presented in Table 3. According to the table, S870 is trivially better than 8800GTS. However, the bandwidth is much worse than that of the main memory to device memory. Fortunately, in our implementation, data transmitted from the main memory is much more than data transmitted to the main memory.

**Table 3. Bandwidth between the device memory to the main memory**

| Matrix size | GeForce 8800GTS (GB/s) | Tesla S870 (GB/s) |
|---|---|---|
| 1000*1000 | 0.23 | 0.20 |
| 2000*2000 | 0.12 | 0.13 |
| 3000*3000 | 0.08 | 0.10 |
| 4000*4000 | 0.06 | 0.08 |

The most important advantage of Tesla S870 is its large global memory. On S870, larger scale matrices can be allocated and experiments prove that the speedup of matrix-vector multiplication on can also achieve 10 to 20 on S870.

## 6. Conclusion and future work

In this paper we present a parallelization of spectral clustering and implement it in both OpenMP on multi-core processors and in CUDA on NVIDIA GeForce 8800GTS. We evaluate its performance on both platforms by a series of experiments. According to the experiments, the scalability and speedup are acceptable on 4-core CPU. The implementation on CUDA also achieves good speedup. We believe that CUDA could contribute greatly to performance enhancement as an accelerator to CPU.

Meanwhile, we also observe that data transmission between the main memory and the device memory in CUDA is not symmetric. Applications which need to transmit a huge amount of data from the device memory to the main memory might not achieve good performance in CUDA.

Our future work will focus on:

Firstly, we will analyze the property of spectral clustering in CUDA further and try to optimize the current program to obtain better performance.

Secondly, the performance of *getAffinityMatrix* remains to be improved. After the optimization of *spectralComputing*, *getAffinityMatrix* might become the bottleneck of the program.

Thirdly, the current implementation in CUDA does not consider the situation when a matrix is too large to be loaded in Geforce 8800GTS's device memory. This might be solved in future.

Finally, we hope that the two implementations can be merged, of which kernel code on CPU is also threaded. In NVIDIA CUDA's sample projects, CUDA can work with OpenMP. With OpenMP threads on CPU, the computation capability of CPU can also be fully utilized. In that case, we can achieve better performance on S870 and make full use of its 4 GPGPUs.

## Reference


[1] Ulrike von Luxburg,"A Tutorial on Spectral Clustering", *Technical Report No. TR-149*, August 2006

[2] NVIDIA CUDA,

http://developer.nvidia.com/object/cuda.html.

[3] OpenMP, http://www.openmp.org

[4] Michalakes, J. and M. Vachharajani, "GPU Acceleration of Numerical Weather Prediction", *Workshop on Large Scale Parallel Processing, IPDPS*, 2007

[5] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig, "Molecular Dynamics Simulations on Commodity GPUs with CUDA", *Lecture Notes in Computer Science* , vol 4873/2007, pp. 185-196, 2007

[6] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, "Scan Primitives for GPU Computing", *Graphics Hardward*, 2007

[7] K. J. Maschhoff and D. C. Sorensen, “A Portable Implementation of ARPACk for Distributed Memory Parallel Architectures”, 1996.

[8] D. CALVETTI y, L. REICHEL z, AND D.C. SORENSEN, “An Implicitly Restarted Lanczos Method For Large Symmetric Eigenvalue Problems”, *Electronic Transactions on Numerical Analysis*, Volume 2, pp. 1-21, March 1994.

[9] Wei Xu, Xin Liu, Yihong Gong, “Document Clustering Based On Non-negtive Matrix Factorization”, *SIGIR’03*, 2003

[10] Andy H. Register, *A Guide to MATLAB Object-Oriented Programming*, Chapman & Hall/CRC, 2007

[11] Ilya Burylov, Michael Chuvelev, Bruce Greer, Grey Henry, Sergey Kuznetsov, and Boris Sabanin, “Intel® Performance Libraries: Multi-Core-Ready Software for Numeric-Intensive Computation”, *Intel Techonlogy Journal*, vol 11, issue 04, 2007

[12] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, Wen-mei W. Hwu. “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA”, *PPoPP ’08,* February, 2008

[13] David Kirk, “NVIDIA CUDA Software and GPU Parallel Computing Architecture”, invited talk, Tsinghua University, March, 2008

[14] Andrew Y. Ng, Michael Jordan, and Yair Weiss, “On Spectral Clustering: Analysis and an algorithm”, *Advances in Neural Information Processing Systems*, volume 14, 2002

[15] NVIDIA, http://developer.nvidia.com/object/cuda.html.

[16] NVIDIA, “NVIDIA CUDA Programming Guide”, http://www.nvidia.com/object/cuda_develop.html

[17] NVIDIA, “CUDA CUBLAS Library”, http://www.nvidia.com/object/cuda_develop.html

[18] D. Calvetti, L. Reichel, and D.C. Sorensen, “A Implicitly Restarted Arnoldi/Lanczos Methods For Large Scale Eigenvalue Calculations”, *Electronic Transactions on Numerical Analysis*, vol 2, pp. 1-21, March 1994

[19] CLUTO, http://glaros.dtc.umn.edu/gkhome/cluto/cluto/download

[20] Intel, “Intel Math Kernel Library Reference Manual”, updated in 2007

[21] M. Szularz, J. Weston, M. Clint and K. Murphy, “A Highly Parallel Explicitly Restarted Lanczos Algorithm”, *Lecture Notes In Computer Science*, vol. 1184, 1996

[22] W. Xiong, J. Li, R.M.M. Chen, S. Qiao, “A fast decomposition of banded symmetric toeplitz matrices for parallel processing”, *Circuits and Systems*, 1999.