SAND2011-4882C

GPU Accelerated Microarray Data Analysis Using Random Matrix Theory

Joey Ingram
Information Engineering Department
Sandia National Laboratories*
Albuquerque, NM 87111, USA
Email: jbingra@sandia.gov

Mengxia Zhu
Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA
Email: mzhu@cs.siu.edu

Frackson Mumba
Department of Curriculum & Instruction
Southern Illinois University
Carbondale, IL 62901, USA
Email: frankson@siu.edu

***Abstract*—Recent advances in high-throughput genomic technology, such as microarrays, usually produce vast amounts of gene expression data under many experimental conditions. Analyzing such data is often difficult due to the colossal data size and the intensive computing involved. In addition, many existing analysis tools often require the inference of experienced analysts and subjective judgments. In this paper, we developed a parallel approach based on Random Matrix Theory (RMT) to generate transcription networks using Graphical Processing Units (GPUs). Recently, GPUs have been redesigned into a more unified architecture, which has allowed them to be used more readily in general purpose computing. This architectural advancement has resulted in GPUs becoming easily programmable parallel processors with performance that is vastly superior to CPUs. Our GPU-based approach makes automated microarray data analysis faster, more accurate and noise resistant without engaging remote high performance computing facilities, such as a cluster or supercomputer. The implementation moves some computationally intensive tasks, such as the calculations of Pearson correlation coefficients, tridiagonal reduction, backtransformation of eigenvectors, and orthogonal rotation, to the GPU. Experimental results on real microarray datasets show that our GPU implementation runs faster than a CPU version using highly optimized LAPACK routines. The runtime speedup gets higher as the number of genes and sample points in a microarray dataset increases.**

I. Introduction

Over the past three decades, there has been a significant increase in genomic research. This increase has led to various developments in high-throughput genomic technologies, including microarrays. A microarray is simply a dataset that contains several hundred sample points for thousands or tens of thousands of genes, which are subject to a noisy environment. These microarrays, along with sequence similarity and chromosomal proximity, can be used to cluster these genes, with genes in the same cluster generally being used for a certain physiological function or being a part of a particular molecular complex. The clustering of such genes can then help guide researchers in such areas as protein-protein interactions and subcellular localizations, which can lead to faster and more accurate genomic research.

However, generating meaningful functional models from genomic data is very challenging due to the nature of biological datasets and algorithmic difficulties for optimality. Researchers have proposed many techniques implementing various methods, which include Boolean networks, Bayesian networks, differential equation-based networks, k-means clustering, hierarchical clustering, self-organizing maps (SOM), and many others. Unfortunately, most of these techniques require researchers to make various decisions during the implementation or construction process that introduces subjectivity (e.g. parameter selection). Some of these techniques are also very sensitive to noise or may not find an optimal global solution. Furthermore, many of these implementations cannot handle large-scale datasets efficiently due to limited available memory and sequential implementation [16].

Nowadays, the GPU is a highly programmable parallel processor with arithmetic and memory performance that is vastly superior to multi-core CPUs [12]. The performance achieved by GPUs is not accomplished using different hardware components as in multi-core CPUs; instead, it is accomplished by taking the inherent design of the GPU to massive scales. A multi-core CPU replicates ALUs, control logic, and execution contexts into independent cores in order to increase throughput. Instruction streams are executed on each core in parallel. Currently, most multi-core CPUs generally have between two and eight cores. In contrast, GPU designs can have upward of thirty cores and increase performance even further by using floating-point ALUs and single instruction multiple data (SIMD) processing. The efficiency of SIMD processing allows the ALUs to be densely packed with many cores. GPUs also use hardware multithreading to hide memory access and instruction fetching latencies. This design allows GPUs to perform hundreds of gigaflops faster than multi-core CPUs [4].

Meanwhile, since the release of high-level programming languages for GPUs, such as CUDA, many researchers have begun using GPUs for general purpose computing (GPGPU). Many higher-level algorithms and applications that have taken advantage of the computational power of the GPU have involved a vast amount of subject areas, such as sorting, searching, database queries, solving differential equations, linear algebra [12], matrix-vector multiplication [5], seismic

* Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

modeling, neural networks, magnetohydrodynamics, signal processing, astrophysics [11], and many others.

This paper proposes the design and implementation to construct and analyze various aspects of transcriptional networks based on RMT [16] using GPUs. No human inference or subjectivity is involved, and a relatively high level of noise, which is inherent to most biological datasets, is removed before clustering is conducted. In our previously published article, our RMT-based approach was applied to two datasets: (i) a yeast cycle microarray dataset [15] with about 2,500 genes and 79 samples, and (ii) a human liver cancer microarray dataset [7] with about 20,000 genes and 207 samples, with good clustering results conforming to previous literature [16]. Thus, the effectiveness of this approach was verified.

However, since our previous parallel RMT program contains some packages that need to be run on a Linux cluster or supercomputer to handle large datasets, it is not convenient and straightforward for ordinary users to deal with the remote HPC facilities. In addition, the computing cycles of a supercomputer are a limited resource and may not be immediately allocated when requested. We believe that GPU-based bioinformatics tools will be a promising and convenient computing paradigm for computational biologists to explore in the future due to its performance, convenience, and cost-effectiveness.

II. GPU Architecture

Historically, GPUs only excelled at three-dimensional graphics processing, and trying to program a GPU for general purpose computing was difficult and time-consuming. This difficulty was a result of the GPU being a fixed-function processor built around a graphics pipeline [12]. This pipeline contained fixed-function stages that performed API-specific operations and programmable stages whose behavior was defined by application code.

Traditionally, these stages were separated due to the vastly different instruction sets, and some aspects were only configurable, not programmable. Another downside was that the programmer did not have direct access to the programmable units, but could only access them at an intermediate step along the graphics pipeline.

Eventually, the stages started to increase in functionality, and their instruction sets started to converge. Therefore, GPU architects decided to change the strict task-parallel pipeline to a more unified architecture, in which all programmable units in the pipeline share a single programmable hardware unit, which allowed the GPU to exploit both task and data parallelism. Since this architecture allowed the programmer direct access to the programmable units, there was a demand for a new programming model. With the old architecture, programmers had to use the graphics APIs even if they were trying to create a program for general purpose computing. With the unified architecture, programmers wanted a higher-level C-like programming language that would abstract the graphics portions of the GPU. This led to languages such as Brook, Microsoft's Accelerator, AMD's HAL and CAL, and NVIDIA's CUDA. The design of the GPU is typically well-suited for applications with the following characteristics [12]:

1) **Computational requirements are large.** Numerous computations must be performed as fast as possible.
2) **Parallelism is substantial.** The application has to be well-suited for parallelism; otherwise, it cannot fully take advantage of the design of the GPU.
3) **Throughput is more important than latency.** The delay of an individual operation is less important than the amount of calculations that can be performed in a given time.

Our random matrix theory approach for clustering genes using microarray data fits these characteristics. Also, since the same operations are performed on hundreds of samples for thousands of genes, the approach also fits the SIMD processing model very well.

A. *Compute Unified Device Architecture (CUDA)*

Recently, NVIDIA introduced the Compute Unified Device Architecture (CUDA), which contains a new parallel programming model and instruction set architecture for NVIDIA's GPUs. CUDA allows programmers to use various high-level programming languages with CUDA extensions, as an interface to application programming. This interface presents a low learning curve to programmers who already know these programming languages. At its core, CUDA consists of a thread group hierarchy, shared memory access, and barrier synchronization, which are exposed to the programmer using a minimal set of language extensions [10]. This provides a level of abstraction that means a compiled CUDA program can run on any number of cores and does not need to know the exact processor count.

CUDA introduces a few new programming concepts with its programming model and instruction set. A kernel is a portion of a CUDA application that is executed N times in parallel by N different CUDA threads. Each thread contains a thread identification number that is accessible through a built-in variable. This variable is a three component vector that can identify a thread in one, two, or three dimensions and allows threads to be grouped into thread blocks [10]. Threads in the same block share the same memory and can communicate and synchronize with one another, while threads in different blocks cannot [5]. Each block is executed independently and can consist of a maximum of 512 threads [10]. However, each block is transparently split into groups of 32 threads called warps. Each of these blocks is assigned to a multiprocessor, which can execute a maximum of eight blocks concurrently [5]. A kernel can be executed by multiple equally-shaped blocks, which can be organized into one or two-dimensional grids [10].

CUDA also adds a few new function and variable qualifiers as part of the instruction extension to the existing high-level programming language. The added function qualifiers are: `global`, `host`, and `device` [8]. The `global` qualifier declares a function as being a kernel, is executed on the device, and is callable from the host only. The `device` qualifier declares that a function is executed on the device and is callable from the device only. The `host` qualifier declares that a function is executed on the host and is callable from the host only. The added variable qualifiers are [10]: `device`, `constant`, and `shared`. The `device` qualifier declares a variable that resides on the device in global memory and can be accessed by all threads. The `constant` qualifier declares a variable that resides in constant memory space and can be accessed by all threads. The `shared` qualifier declares a variable that resides in the memory space of a thread block and is only accessible by threads in that block. There are also various built-in vector primitive types and variables that are available to the programmer [10].

Many libraries are implemented on top of CUDA. One such library is an implementation of the Basic Linear Algebra Subroutines (BLAS), which is known as the CUBLAS library [3]. This library provides various generic linear algebra functions to perform basic matrix-vector and matrix-matrix operations [9].

B. Limitations of GPUs and the CUDA Architecture

However, there are a few limitations of GPUs for programming purposes. GPUs have a limited amount of memory. If a very large dataset cannot fit into GPU memory at once, the computation would have to be split up over the dataset and each section of the dataset would have to be swapped into memory. Significant communication overhead would be introduced by such memory swapping, which will degrade the performance of the GPU [8].

A disadvantage of CUDA is that threads in different blocks cannot communicate. Although this improves performance by allowing blocks to execute in any order, the lack of communication means that these threads cannot synchronize, which must be considered by the programmer during algorithm development. Also, in the CUDA architecture, if two addresses of a memory request by threads in the same half-warp fall into the same shared memory bank, there is a bank conflict and the access must be serialized. However, no bank conflict occurs when threads of a half-warp read from the same 32-bit word [5]. Therefore, it is beneficial to ensure that the data is aligned in such a way to reduce these bank conflicts.

III. Technical Approaches

A. Random Matrix Theory Approach

We proposed an approach that uses RMT to construct and analyze transcription networks from microarray data. The RMT algorithm lends itself well to parallelization. The approach can be summarized in the following steps [16]:

1) **Normalize the data.** The expression signal of gene $i = 1, \ldots, N$ for sample $s = 1, \ldots, K$ is defined as:

$$W_i(s) \;\equiv\; \ln\left(\frac{Es_i(s)}{Ec_i(s)}\right) \;, \tag{1}$$

where $Es_i(s)$ is the expression signal of samples for gene i and $Ec_i(s)$ is the corresponding control signal. The data is then normalized as:

$$w_i(s) \;\equiv\; \frac{W_i(s) \;-\; \langle W_i \rangle}{\sigma_i} \;, \tag{2}$$

where σ_i represents the standard deviation of W_i and $\langle W_i \rangle$ stands for the average over different samples for gene i. This normalization of the data will account for the various differences that may exist in the levels of the expression signal exhibited by different genes.

2) **Calculate the correlation between each of the genes.** From the $N \times K$ normalized data, an $N \times N$ correlation matrix is constructed using the Pearson correlation coefficient. The Pearson correlation coefficient C_{xy} between genes x and y is defined as:

$$C_{xy} \;=\; \frac{\sum_{i=1}^{K}(x_i - \bar{x})(y_i - \bar{y})}{(K-1)s_x s_y} \;, \tag{3}$$

where s_x and s_y are the standard deviations of gene x and gene y, respectively. A negative correlation coefficient denotes anti-correlation, a positive correlation coefficient denotes correlation, and a correlation coefficient of zero denotes that no correlation exists.

3) **Remove measurement noise using deviating eigenvalues.** Certain properties of microarrays can introduce noise into the correlation matrix. The correlation between certain genes can fluctuate over time or under different sampling conditions. Also, noise can be introduced due to the limited number of sample points. In order to remove this noise and determine more accurate correlation, the eigenstates of the correlation matrix are compared to those of a truly random correlation matrix. Statistical properties that are similar to both matrices can be interpreted as noise, while differences can be interpreted as true correlation. First, the eigenvalues are calculated for both matrices. Then, the eigenvalues are arranged in ascending order and the probability distributions are compared. The K eigenvalues of the correlation matrix that deviate from the eigenvalue range of the random matrix are kept and considered as genuine correlation.

4) **Transform the components of the corresponding deviating eigenvectors.** After locating the deviating eigenvalues, their corresponding eigenvectors are evaluated. There are N eigenvectors, each with N components corresponding to the N gene variables. The eigenvectors are perpendicular to each other and are normalized to one. The probability distribution of the eigenvector components are compared to that of the random matrix, which follows a Gaussian distribution with zero mean and unit variance. The deviating eigenvector components should therefore have a significant deviation from the Gaussian distribution. After normalizing the eigenvectors, their components are multiplied by the square root of the corresponding eigenvalue to obtain a loading factor.

5) **Cluster using the loading factors and orthogonal (perpendicular) rotation.** Using the computed loading factors, the genes can be clustered. Each eigenvector represents one factor that leads to a gene cluster. A large loading factor indicates that the gene is more expression-dominating for that cluster. In order to make the interpretation of gene clusters easier and more reliable, orthogonal rotation is applied to the retained eigenvectors. A method known as Varimax is used for the orthogonal rotation. A rotation matrix R is defined as:

$$R = \begin{bmatrix} \cos\theta_{i,i} & \cos\theta_{i,j} \\ \cos\theta_{j,i} & \cos\theta_{j,j} \end{bmatrix}, \qquad (4)$$

where $\theta_{i,j}$ is the rotation angle from old axis i to new axis j. After rotation, each eigenvector will contain a small number of large loadings and a large number of small loadings, which means that the gene clusters will consist of a reduced number of dominant genes compared to those without rotation. This ensures that each gene will only load heavily on a very small number of gene clusters.

This approach does not require human involvement and can remove a high level of noise that is common for biological data. It also allows gene clusters to overlap in terms of gene membership. This overlapping of clusters agree with the biological perspective that a single gene may be involved in multiple pathways. Additional calculations can be performed to evaluate the stability and quality of cluster membership, if desired [16]. It should be noted that these steps are just a summary of the overall process, and if the dataset contains time-series gene expression data, then a different similarity metric and correlation calculation must be used.

B. Computation of Eigenstates for Symmetric Matrices

Computing the eigenvalues and corresponding eigenvectors of a full symmetric matrix can be quite computationally expensive. Therefore, algorithms that find the eigenstates of symmetric matrices generally all consist of three phases: reduction of the original dense matrix to a condensed form by orthogonal transformations, solution of the condensed form, and backtransformation of the solution of the condensed form to the solution of the original symmetric matrix. Finding the solution of the condensed form is generally substantially faster than finding the solution to the original matrix. Also, the backtransformation phase is only necessary if the eigenvectors are needed, as the eigenvalues of both the original matrix and the condensed matrix are the same [1].

A popular reduction method used by many algorithms is to reduce the original matrix to tridiagonal form. A tridiagonal matrix is a matrix that only has nonzero elements in the main diagonal, the diagonal below the main diagonal (the subdiagonal), and the diagonal above the main diagonal (the superdiagonal). An example may have the form:

$$\begin{bmatrix} x_{1,1} & x_{1,2} & 0 & 0 & 0 \\ x_{2,1} & x_{2,2} & x_{2,3} & 0 & 0 \\ 0 & x_{3,2} & x_{3,3} & x_{3,4} & 0 \\ 0 & 0 & x_{4,3} & x_{4,4} & x_{4,5} \\ 0 & 0 & 0 & x_{5,4} & x_{5,5} \end{bmatrix}$$

The reduction of a dense, symmetric matrix A to such a form can be accomplished by applying a series of Householder transformations to A. This method can reduce an $n \times n$ symmetric matrix to tridiagonal form by using $n-2$ orthogonal transformations. For each of the first $n-2$ columns of the matrix A, a vector is constructed that reduces the required portion of a whole column and whole corresponding row to zero.

If the eigenvectors of the tridiagonal matrix are found, they can be backtransformed to those of the original matrix by applying the transformations on the eigenvector matrix in the reverse order [14]. This reduction requires a theoretical complexity of $O(\frac{4}{3}n^3 + n^2)$, but the resulting eigenvalue computation is generally much faster when compared to finding the eigenvalues of the original matrix [1].

Currently, there are no functions provided by either CUDA or CUBLAS for computing the eigenvalues and eigenvectors of symmetric matrices [9], [10], which is the most computationally intensive portion of RMT. However, there is a commercially-available library (CULA) that implements a few of the LAPACK functions for symmetric eigendecomposition using the CUDA architecture [6].

IV. Implementation Details

The sequential implementation and CPU portions of the GPU implementation were written in C++ and both were run on an Intel Pentium 4 3.00 GHz machine with 2 GB of RAM running Linux. The GPU used was an NVIDIA Quadro FX 5600, which has 1.5 GB of memory, 16 multiprocessors, 128 cores, and a clock rate of 1.35 GHz.

A. CPU Implementation

For the sequential version, after loading the dataset, a random data matrix is constructed, where each element of the random matrix is contained in the range of the minimum and maximum element of the raw data matrix. After the Pearson correlation matrix is constructed for both the raw data and random matrices, both Pearson matrices are reduced to tridiagonal form using the LAPACK subroutine `SSYTRD`, which utilizes the Householder reduction method. Then, the eigenvalues of both matrices are computed using `SSTEBZ`, which uses the bisection method. This subroutine outputs the eigenvalues in ascending order, which means no sorting is necessary [1].

After computing the eigenvalues for both matrices, the eigenvalues of the raw data Pearson correlation matrix is compared to the largest eigenvalue of the random data Pearson correlation matrix to determine the K meaningful eigenvalues. Then, the corresponding eigenvectors of the K meaningful eigenvalues of the raw data Pearson matrix are computed using the `SSTEIN` subroutine, which utilizes inverse iteration [1]. Since K is generally relatively small (usually less than 30) and only these eigenvectors are necessary for analysis, this method is much faster than computing all of the eigenvectors. However, the eigenvectors obtained from `SSTEIN` are those of the tridiagonal matrix and not the original symmetric matrix. Therefore, the eigenvectors must be transformed to those of the original matrix by using the elementary reflectors and scalar factors obtained from running `SSYTRD`, which can be accomplished by the `SORMTR` subroutine [1]. After the K eigenvalues and their corresponding eigenvectors are obtained, the eigenvectors are transformed to component loadings by multiplying each eigenvector by the square root of its corresponding eigenvalue. Then, these loadings are rotated using the Varimax rotation method. After rotation, the resulting matrix is used to cluster the genes into groups (a threshold of 0.55 was used).

As noted in Section III-B, the tridiagonal reduction phase requires $O(\frac{4}{3}n^3+n^2)$, but the resulting eigenvalue computation is generally quite fast [1]. Therefore, the tridiagonal reduction phase and subsequent backtransformation are the main bottlenecks of the CPU implementation.

B. GPU Implementation

For the GPU implementation, after loading the dataset and generating the random data matrix, both matrices are transferred to device memory. The Pearson correlation matrix of the random data matrix is then computed. The random Pearson matrix is then reduced to tridiagonal form using custom kernels and functions provided by CUBLAS. Block updates are performed using the cublasSsyr2k method [9]. The diagonal and off-diagonal elements of the resulting matrix are then transferred back to host memory. After that, the raw data Pearson correlation matrix is computed and reduced to tridiagonal form. The reduction vectors used are stored so that the tridiagonal eigenvectors can be backtransformed to those of the original symmetric matrix. By performing the steps in this order, the amount of memory used is reduced, because the only elements needed to compute the eigenvalues of the random data Pearson matrix using bisection are the diagonal and off-diagonal elements of the tridiagonal matrix. After both Pearson matrices are reduced to tridiagonal form and the diagonal and off-diagonal elements are transferred back to host memory, the eigenvalues of these matrices are computed using SSTEBZ. Again, the K meaningful eigenvalues are found, and the corresponding eigenvectors are computed using SSTEIN. The K meaningful eigenvalues and corresponding eigenvectors are then transferred back to device memory, where the eigenvectors are backtransformed using the stored transformation vectors and a few basic CUBLAS functions (cublasSgemv and cublasSger). The transformation of the eigenvectors to component loadings and Varimax rotation are also done on the GPU. The results are then transferred back to host memory, so that the genes can be clustered.

The main speedup obtained by the current GPU implementation results from the reduction of the symmetric matrices to tridiagonal form and the backtransformation of the eigenvectors, which seem to be the main bottlenecks of the sequential version. As a matter of fact, the current GPU implementation requires many device-to-host and host-to-device memory copies, which would be eliminated if a mature linear algebra library were available for the CUDA architecture.

V. Experimental Results

Table I shows the results obtained from running both the pure CPU and GPU implementations on seven different microarray datasets with a varying number of genes and sample points. Five of the datasets were obtained from ArrayExpress, a database of gene expression and other microarray data maintained by the European Bioinformatics Institute [2]. The ArrayExpress identification number is given for datasets that were obtained from this database, for future reference. The clustering results conform to those in previously published articles. Figure 1 illustrates a visual transcription network generated by the GPU implementation for the human gastric cancer cells dataset (created using Pajek [13]). The vertices denote genes and the edges represent expression correlation between genes. Genes belonging to the same cluster are similarly colored and placed close to each other. Both the CPU and GPU implementation analyzed each dataset ten times. The elapsed time for each run was then averaged to obtain a more accurate running time. For smaller matrices, the GPU implementation performs similarly to the sequential implementation or slightly worse. In this case, the memory latency of the GPU and performing the various host-to-device and device-to-host memory copies outweighs the benefit obtained from the efficiency of the GPU. In general, the GPU implementation only provides a total average speedup of approximately 1.16. However, the average speedup for larger matrices (with more than 2000 genes) is approximately 1.44. Therefore, the benefits of using the GPU are only seen when the number of genes is fairly large.

VI. Conclusion and Future Work

The Pearson correlation matrix calculated from the microarray data typically contains both genuine and random components. The random component is removed by testing the statistics of the eigenvalues of the correlation matrix against the eigenvalues of a truly random correlation matrix obtained from a mutually uncorrelated expression data series. The investigation into the components of deviating eigenvectors reveals distinct functional modules.

Our RMT method has many advantages over existing clustering methods. For example, the number of functional modules can be automatically inferred and a single gene can be grouped into multiple clusters; it is essentially a global clustering method as opposed to some local clustering method used by greedy algorithms. The utilization of GPUs reduces the amount of computation time that is otherwise needed on a pure sequential program.

TABLE I
RESULTS OBTAINED FROM VARIOUS DATASETS.

Dataset	Number of genes	Sample points	ArrayExpress ID	CPU Time (in seconds)	GPU Time (in seconds)
Test	9	–	–	0.1	0.1
Human gastric cancer cells	315	353	E-TABM-341	0.8	1.3
Mouse epididymal adipocytes	574	4	E-MEXP-1932	2.4	2.5
Chorioamniotic membranes	1491	30	E-TABM-469	23.1	17.5
Human cancer samples	1866	22	E-TABM-184	39.8	29.3
Drosophila abdomen samples	2340	16	E-MEXP-2080	70.5	49.1
E. coli samples	4345	445	–	396.8	274.9

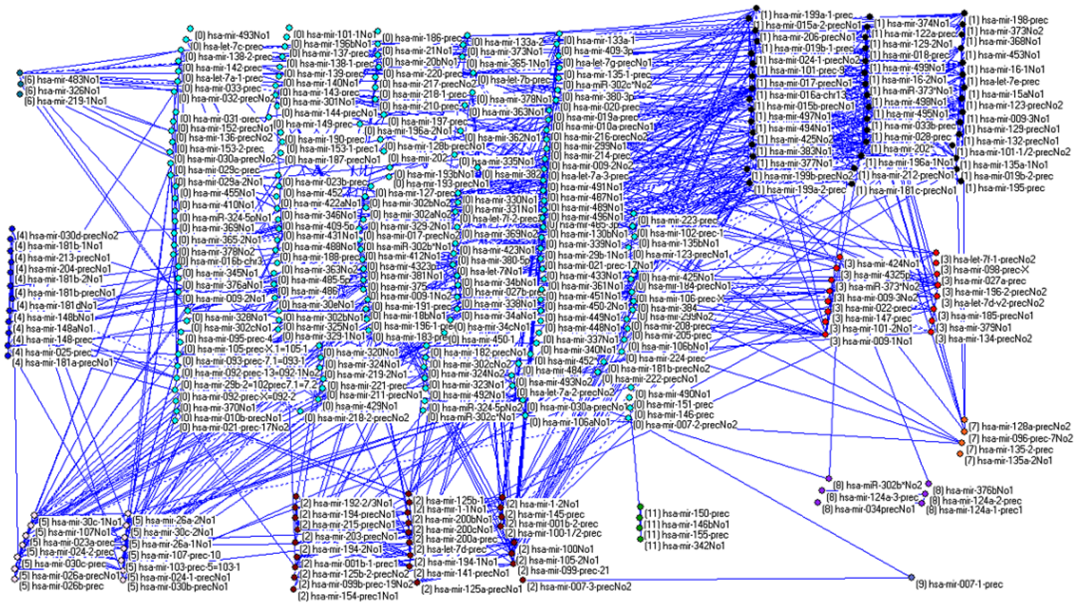

Fig. 1. Transcription Network of Human Gastric Cancer Cells Dataset (E-TABM-341) generated by Pajek.

There is still much work to be done. Optimized functions for computing eigenvalues and eigenvectors of a symmetric matrix using CUDA or CUBLAS needs to be developed. This would reduce the amount of device-to-host and host-to-device transfers in the current implementation, which in turn would hopefully increase the speed of the GPU implementation.

VII. ACKNOWLEDGEMENTS

The authors would like to thank the Illinois State Board of Education for support under the title of ”Partnership for Improved Science Achievement through Computational Science (PIASCS)” funded in 2010. We plan to use work from this project as part of the teaching materials in our PIASCS project workshops.

REFERENCES

[1] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D.: LAPACK Users Guide, 3rd ed., SIAM, Philadelphia (1999)

[2] ArrayExpress, http://www.ebi.ac.uk/microarray-as/ae/

[3] Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Orti, E.S.: Evaluation and Tuning of the Level 3 CUBLAS for Graphics Processors. In: 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing, pp. 1-8. (2008)

[4] Fatahalian, K. and Houston, M. A Closer Look at GPUs. Comm. of the ACM, vol. 51, no. 10., pp. 50-57, (2008)

[5] Fujimoto, N.: Faster Matrix-Vector Multiplication on GeForce 8800GTX. In: 22nd IEEE International Parallel and Distributed Processing Symposium, IEEE Press, (2008)

[6] Humphrey, J.R., Price, D. K., Spagnoli, K. E., Paolini, A. L., Kelmelis E. J., ”CULA: Hybrid GPU Accelerated Linear Algebra Routines,” SPIE Defense and Security Symposium (DSS), April, 2010.

[7] Liver Microarray Dataset, http://smd.stanford.edu/cgi-bin/publication/viewPublication.pl?pub_no=107

[8] Muller, C., Frey, S., Strengert, M., Dachsbacher, C., Ertl, T.: A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality. IEEE Transactions on Visualizations and Computer Graphics, vol. 15, no. 4, (2009)

[9] NVIDIA, CUDA CUBLAS Library 2.0. http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf, (2008)

[10] NVIDIA, NVIDIA CUDA Programming Guide 2.3.1. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf, (2009)

[11] NVIDIA CUDA Zone, http://www.nvidia.com/object/cuda_home.html

[12] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU Computing. In: Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899. (2008)

[13] Pajek. http://vlado.fmf.uni-lj.si/pub/networks/pajek/

[14] Press, W.H., Flannery, B.P., Teukolsky S.A., Vetterling, W.T.: Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, (1992)

[15] Yeast Microarray Dataset, http://genome-www.stanford.edu/cellcycle

[16] Zhu, M., Wu, Q.: Transcription Network Construction for Large-scale Microarray Datasets using a High-performance Computing Approach. BMC Genomics, vol. 9 suppl. 1, (2008)