Speeding up the MATLAB complex networks package using graphic processors*

Zhang Bai-Da(张百达)[a)†], Tang Yu-Hua(唐玉华)[b)],
Wu Jun-Jie(吴俊杰)[a)], and Li Xin(李 鑫)[a)]

[a)] *National laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha* 410073, *China*

[b)] *Department of Computer Science and Technology, School of Computer, National University of Defense Technology, Changsha* 410073, *China*

(Received 18 March 2011; revised manuscript received 4 May 2011)

The availability of computers and communication networks allows us to gather and analyse data on a far larger scale than previously. At present, it is believed that statistics is a suitable method to analyse networks with millions, or more, of vertices. The MATLAB language, with its mass of statistical functions, is a good choice to rapidly realize an algorithm prototype of complex networks. The performance of the MATLAB codes can be further improved by using graphic processor units (GPU). This paper presents the strategies and performance of the GPU implementation of a complex networks package, and the Jacket toolbox of MATLAB is used. Compared with some commercially available CPU implementations, GPU can achieve a speedup of, on average, 11.3×. The experimental result proves that the GPU platform combined with the MATLAB language is a good combination for complex network research.

Keywords: complex networks, graphic processors unit, MATLAB, Jacket Toolbox

PACS: 89.75.Hc, 87.23.Ge, 89.20.Hh, 89.75.–k **DOI:** 10.1088/1674-1056/20/9/098901

1. Introduction

Network ideas have been applied successfully in many fields such as the Internet and the world wide web, epidemiology, scientific citation and collaboration, metabolism, and ecosystems.[1−4] Recently, however, a substantial new movement appeared in network research, with the focus shifting from the analysis of single small graphs and the properties of individual vertices or edges within such graphs to the consideration of large-scale statistical properties of graphs.[5,6] This new trend has been driven largely by the availability of computers and communication networks that allow us to gather and analyse data on a scale far larger than previously.[6−8] Therefore, speeding up the common algorithms of complex networks is exigent.

The graphic processor unit (GPU), first invented by NVIDIA[9] in 1999, is the most pervasive parallel processor to date. Fueled by the insatiable desire for life-like real-time graphics, the GPU has evolved into a processor with unprecedented floating-point performance and programmability. Today's GPU greatly outpaces CPU in arithmetic throughput and memory bandwidth, making them the ideal processors to accelerate a variety of data parallel applications. Attempts to utilize GPU not only for manipulating computer graphics, but also for different purposes, led to a new research field, called "general-purpose computation on GPU (GPGPU)". In 2006, NVIDIA introduced compute unified device architecture (CUDA), a general purpose parallel computing architecture, which leverages the parallel compute engine in NVIDIA GPUs to model and solve many complex computational problems in a more efficient way than on a CPU. The CUDA enables the GPU to be programmed with a variety of high level programming languages, such as C, C++, Fortran, OpenCL, DirectCompute and so on.[9] Although the GPU can accelerate algorithms of complex networks, it is not realistic to expect the average programmer to deal with all the complexity introduced by CUDA programming. Therefore, an in-

*Project supported by the Science Fund for Creative Research Groups of the National Natural Science Foundation of China (Grant No. 60921062), the National Natural Science Foundation of China (Grant No. 60873014), and the Young Scientists Fund of the National Natural Science Foundation of China (Grant Nos. 61003082 and 60903059).

†Corresponding author. E-mail: zhangbaida@gmail.com

© 2011 Chinese Physical Society and IOP Publishing Ltd

terface which is easier to use is needed.

MATLAB is a high-level technical computing language which can be used to solve technical computing problems faster than traditional programming languages, such as C and C++. However, its efficiency was low in comparison with the C language. The Jacket MATLAB Toolbox developed by AccelerEyes LLC (www.accelereyes.com) is a toolbox designed to run MATLAB routines on the GPU.[10] Jacket enables programmers to write and run codes on the GPU in the native M language used in MATLAB. The MATLAB functions are transformed into GPU functions by casting input data to Jacket's GPU data structure. The interpretive nature of the M language is maintained by providing real-time, transparent access to the GPU compiler. With the Jacket Toolbox and GPU hardware, we can run the MATLAB codes in less time. This paper presents an implementation of several algorithms of complex networks using the Jacket Toolbox under the GPU platform and gives some guidelines on the use of GPU using the MATLAB language.

To sum up, the contributions of this paper are as follows.

1) A solution is presented to improve the performance of complex networks algorithms.

2) Several examples of GPUlizing[1)] M-code and some general advice about M-code GPUlizing is given.

3) It is validated that M-code running on GPU can greatly improve the performance in comparison with its CPU counterpart.

The rest of this paper is organized as follows. Section 2 describes the motivation for using GPU to speedup algorithms of complex networks. Section 3 is about how the M-code is GPUlized efficiently. Section 4 presents the experimental results. Section 5 briefly summarizes the related works. Finally, Section 6 concludes this paper. To simplify the discussion, when results of GPU computing are mentioned in what follows, it means that GPU and CPU work together.

2. Motivation

GPU has now become a powerful device for the execution of data-parallel, arithmetic (versus memory) intensive applications in which the same operations are carried out on many data elements in parallel. The GPU is different from CPU in that the number of cores in a GPU is large and the architecture of the core is simple. Therefore, the GPU is very suitable for work having a large number of parallelism. In the area of complex networks, statistical mechanics was broadly used, and there is a lot of work which can be done at all vertices in parallel. Tasks like this are very suitable for GPU to compute. It is very natural to map some vertices and edges to parallel processing threads. For example, the degree distribution of a graph having 10000 vertices is needed, and there are 10000 threads which can be executed in parallel. The CPU is designed for speeding up serial codes, so work which cannot be parallelized is the very work which can be better done by CPU. When the programme is executed, it can be speeded up by executing the parallel part on GPU and the serial part on CPU. Tianhe-1A, the fastest supercomputer in the world, has heterogeneous architecture which has CPU and GPU nodes. Therefore, if GPU and CPU are both utilized, it will be easy to transform the code to supercomputer when some larger networks need to be analysed.

2.1. Complex computing

Because of information technology, very large scale networks can be studied now. However, some problems still exist. The number of vertices is so large that anything done on a network will cost a lot of time that we do not have. Taking the human neural network as an example, there are more than 10^{10} neurons, even the degree distribution is difficult to compute. We need a platform that can execute the operation more quickly but with a cost that is in the range that most the groups can accept. The memory space needed for storing a network is proportional to the network size. If an adjacent matrix is used, all pairs of vertices need a logical value to indicate whether there is an edge, then memory space complexity is $O(N^2)$, where N is the total number of vertices contained in the network. If a listed structure is used to store a network, each edge needs two numbers to indicate its two ends, then memory space complexity is $O(E)$, where E is the total number of edges contained in the network. Suppose 4-GB memory space is available, only about 30000 network vertices can be stored when an adjacent matrix is used, and about 10^9 network edges can be stored when a listed structure is used. When time complexity is considered, it is expressed by the function of problem size. Taking the average path length algorithm as an example, its time complexity is $O(NM)$, where N and M are the

[1)] In this paper, "GPUlize" means transforming M-code into hybrid M-code that can run on GPU

numbers of vertices and edges respectively. Suppose a network with 10000 vertices and 100000 edges, this $N \times M$ is 10^9. Therefore, some hardware and software are needed to implement the algorithm more quickly. Parallel computing is the very technology. A parallel computer divides the problem into several small ones, and assigns each small problem to a processor. And all processors collaborate to solve the big problem.

2.2. Speeding up computing with GPU

A heterogeneous system with a GPU and a CPU are shown in Fig. 1. The GPU consists of a number of streaming multiprocessors (SM).[11] Each one includes 32 shader processor (SP) cores, a local memory shared by all SPs, 32768 registers, 2 super function units (SFU), and fast ALU units for hardware acceleration of transcendental functions. A global memory shared by all SMs provides capacity up to 4 GB and memory bandwidth up to 144 GB/s. The shared memory, which is a small area of on-chip memory (currently 16 kB per multiprocessor) that can be accessed in parallel by blocks of threads, is the most significant part. This allows caching frequently used data and can provide high speedup by using textures to access data. Combined with a thread synchronization primitive, this allows cooperative parallel processing of on-chip data, thus greatly reducing the expensive off-chip bandwidth requirements of many parallel algorithms. The research into complex networks can benefit greatly from the high level parallelism.

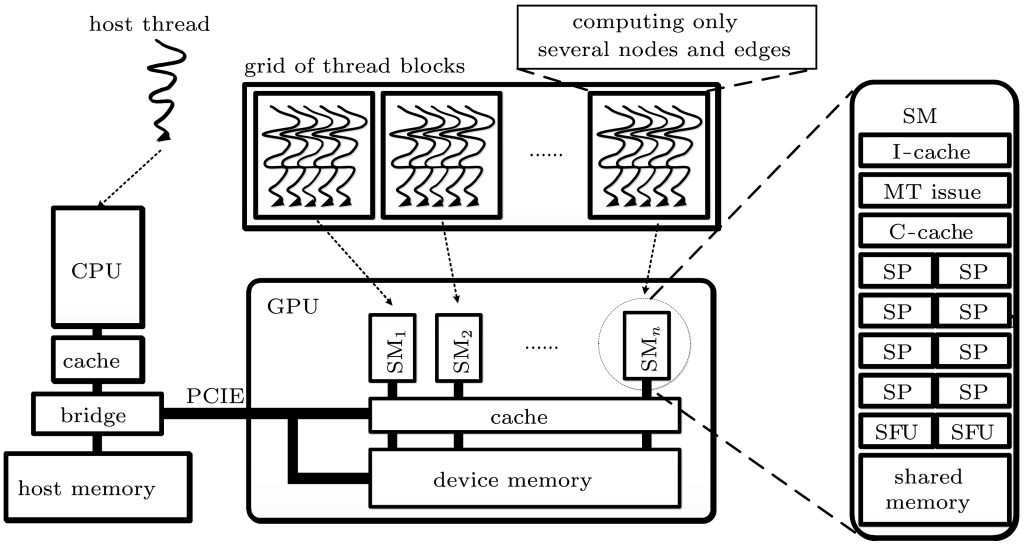


Fig. 1. Architecture of GPU & CPU System.

As we know, the CPU excels at executing a few serial threads. When sequential tasks are executed, memory accesses are all low latency cached memory accesses, and the CPU performs well.[12] However, when there are many threads executed in parallel and high bandwidth memory access is needed, the GPU performs better. In the study of complex networks, the task is the very thing for the GPU, the features of each vertex need to be computed, and very high bandwidth is needed. There are many papers which focus on how to utilize GPU resource by CUDA programming, but few on how to utilize GPU by MATLAB, a more advanced language. It has been calculated that the size of MATLAB codes is only 1/7 to C codes for CPU when the same function is implemented.

3. Methods and strategies

With the release of NVIDIA's CUDA, the hardware architecture of modern GPUs can be viewed as a series of Single Instruction Multiple Data (SIMD) multiprocessors, each is capable of processing a set of instructions on different memory elements in a clock cycle. Once a GPU data structure has been created, any operation on that GPU matrix is performed on the GPU rather than on the CPU. To turn off GPU computation, it can simply cast the data back to the CPU

using the function provided by the Jacket Toolbox. Any function called on GPU data will be executed on the GPU automatically without any extra programming. The GPU offers the best performance when all its processing cores and memory are maximally utilized. Therefore, from the user's perspective, an analysis of work is the best way to determine which jobs should be loaded on the GPU. Some strategies and examples for GPUlizing the M-code are listed below.

3.1. Assigning tasks properly

The GPU speedup benefits from higher global memory bandwidth and floating-point operation performance, hardware scatter-gather support, and acceleration of transcendental (e.g. exponential and trigonometric) functions. However, if the memory transfer between CPU and GPU arises frequently, the performance will degrade badly. In Fig. 2, the cost of memory transfer between CPU and GPU is shown by plotting the transfer time as a function of the matrix size. Thus, it is better to define the data structure in GPU memory directly, when GPU and CPU work together. Although it is always possible to use GPUsingle or GPUdouble functions to create a GPU variable, these functions perform a memory transfer from CPU memory to GPU memory (they copy the content of the CPU memory to the GPU memory). It is faster if the GPU array is directly created on the GPU memory. For example, if a random network is needed, there are two methods which can be used to generate a GPU matrix. One is generating a matrix directly on GPU memory, the other is copying a CPU array to GPU memory. The M-code of those two methods is shown in Fig. 3, in which the connecting probability of a random graph is supposed to be 0.2. The best way to reduce the cost for data transfer is assembling the GPU computing together.

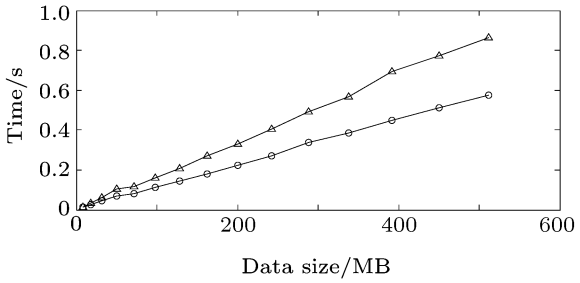


Fig. 2. Transfer time between CPU and GPU.

```
graph = grand(N)<0.2;
```

(a) get a GPU matrix directly

```
tmp = rand(N)<0.2;
graph = glogical(tmp);
```

(b) get a GPU matrix indirectly

Fig. 3. Two ways to generate a random graph in GPU memory.

3.2. Exploiting more parallelism

There is a sentence in Jacket called *gfor* that can run the *for* loop in parallel. If you want to further improve the performance, you can rewrite your M-code by simply replacing the *for* sentence by *gfor*. The *gfor*/*gend* loop construct can launch all of the iterations of a *for* loop on the GPU simultaneously, as long as the iterations are independent. For example, if degree distribution is needed, the M-code can be rewritten as shown in Fig. 4. The script achieves a 728% speedup on a dual-core Intel machine with an NVIDIA GTX 480 GPU.

```
gfor  k = 1:N  %N is the number  of vetex
      %graph is the adjacent matrix
      degree(i) = sum( graph(k ,: ))
gend
```

Fig. 4. The M-code for counting degree of a graph.

3.3. Matrix calculation preferred

MATLAB is optimized for matrix calculation, and the Jacket Toolbox is also suitable for matrix calculation. Therefore, only mapping the *for* loop to matrix operation will obtain a great performance improvement. Taking the script in Fig. 4 as an example, the code can be rewritten as only one sentence as

```
>> degree=sum(ggraph);
```

and 437% speedup can be achieved on a dual-core Intel machine with an NVIDIA GTX 480 GPU.

4. Experiment

4.1. Experimental setup

In order to evaluate the speedup of the M-code running on GPU in comparison with the CPU counterpart, the NVIDIA GPU machine (currently, Jacket makes use of CUDA which is only implemented for the NVIDIA cards) is used. The version of MATLAB

is V7.11.0(R2010b). The Jacket Toolbox is obtained from the Accelereyes web site, version V1.6.1 (build 328). The CPU is Intel Core 2 Quad Q9550, equipped with 4 GB memory. The GPU and Jacket information is shown in Fig. 5, which is obtained by the Jacket *ginfo* function. Before the Jacket Toolbox can work correctly, the following steps are needed in advance:

1) setup MATLAB software;

2) setup NVIDIA CUDA drivers;

3) setup Microsoft Visual C++ 2008 Redistributable Package and Microsoft. NET framework;

4) setup Jacket Toolbox (Version 1.6.1).

For a more detailed description of the software setup, please refer to www.accelereyes.com.

```
Jacket v1.6.1 (build 10299) by AccelerEyes
License Type: Designated Computer (on 64-bit Windows)
Licensed Addons: SDK, JMC, DLA

Detected CUDA-capable GPUs:
CUDA driver 263.06, CUDA toolkit 3.1
GPU0 GeForce GTX 480,810 MHz, 1504 MB VRAM,
Compute 2.0 (single, double)
Display Device: GPU0 GeForce GTX 480
GPU Memory Usage: 1349 MB free (1504 MB total)
```

Fig. 5. GPU and Jacket Toolbox info.

4.2. Benchmarks

The following is a short description of the benchmarks, and how to GPUlize them. These benchmarks are used to test our method in the next section.

4.2.1. Count degree

Count degree is used for counting the degree of all the vertices in a network. Because each vertex has a line in an adjacent matrix, this is very suitable for parallel counting. We can use the GPU function *sum* to obtain the result quickly.

In the MATLAB environment, this can be done as follows:

```
% generate a GPU matrix
>> gadjMatrix = gsingle(adjMatrix);
% count the edges of every vertex
>> Degree = sum(gadjMatrix);
```

4.2.2. Generation of random networks

This benchmark is used to generate a random network used for testing other algorithms. The main part of the algorithm is generating the probability matrix and setting the adjacent matrix elements according to the probability. We can use the *grand* function of Jacket to generate the probability matrix on GPU memory concurrently, and set each element by comparing with the predefined probability threshold. The M-code can be written as follows:

```
% generate a random matrix in GPU memory
>> data = grand(N);
% where p is the probability of connection
>> gadjMat = data>p;
% made gadjMat symmetrical
>> gadjMat = triu (gadjMat,1) +triu (gadjMat,1)' + eye(N);
```

4.2.3. Average path length

As we know, the average path length (APL) is computed between every pair of vertices. This can be done in parallel by using the *gfor* sentence very efficiently. And it is calculated by

$$APL = \frac{2}{N(N-1)} \cdot \sum_{i=1}^{N} \sum_{j=i}^{N} spl_{ij}, \qquad (1)$$

where spl_{ij} is the shortest path length between vertex i and vertex j.

In MATLAB environment, this can be done as follows:

```
>> gspl = -1*gones(N);
>> mask = gones(N);
>> for i=1:N
>>     mask = gspl < 0;
>>     tmp = tmp * gadj;
>>     index = tmp .* mask > 0;
>>     gspl(mask) = i;
>> end;
>> gspl = adj*adj;
% gotten APL from spl matrix
>> apl = sum(sum(gspl));
```

4.2.4. Clustering coefficient

The clustering coefficient is a feature describing the cluster degree of a network. It is calculated by

$$C = \frac{1}{N} \sum C_i, \quad C_i = \frac{2e_i}{x_i(x_i - 1)}, \qquad (2)$$

where e_i is the number of edges existing in the neighbour of vertex i, x_i is the degree of vertex i, and N is the number of vertices in network.

In the MATLAB environment, this can be done as follows:

```
>> gadj = gsingle(adj);
% get the degree of all vertices
>> degree = sum(gajd);
>> gfor i = 1:N
>>    index = find(gadj(i,:));
>>    ndLclAdj = gadj(index,index);
>>    lclNumEdge(i) = cnnz(ndLclAdj)/2;
>> gend
>> C = 1/N*sum(lclNumEdge./(degree.*(degree-1)));
```

4.2.5. Degree correlations

Newman presented that degree correlations were simply the Pearson correlation coefficients of the degrees at either end of an edge and the value is in the range $-1 \leq r \leq 1$. The expression of degree correlations can be written as follows:[13,14]

$$r = \frac{M^{-1}\sum_i j_i k_i - [M^{-1}\sum_i \frac{1}{2}(j_i + k_i)]^2}{M^{-1}\sum_i \frac{1}{2}(j_i^2 + k_i^2) - [M^{-1}\sum_i \frac{1}{2}(j_i + k_i)]^2}, \quad (3)$$

where M is the number of edges, j_i, k_i are the degrees of the vertices at the ends of the i-th edge, with $i = 1, \cdots, M$.

In the MATLAB environment, degree correlations can be computed on GPU by using the following code:

```
% transfer adj matrix for CPU memory to GPU memory
>> gadj = gsingle(adj);
% get all non zero elements
>> [rows,cols,vals] = find(gadj==0);
% get the degree of all vertices
>> degree = sum(gadj);
% get two ends degree of all edges
>> ji = degree(rows(:)); ki = degree(cols(:));
>> tmp1 = sum(ji.*ki) / M;
>> tmp2 = (sum(ji+ki)/2/M)^2;
>> tmp3 = sum(ji.^2 + ki.^2)/2/M;
% Degree Correlations
>> r = (tmp1-tmp2)/(tmp3-tmp2);
```

4.2.6. Modularity

Modularity is a metric first proposed by Newman for measuring the strength of the community structure of a given partition.[3] Considering a particular community which divides a network into k communities, we can write the expression as follows, in which, f_{ij} is the fraction of all edges in the network that links vertices in community i to vertices in community j, and a_i is the sum of f_{ij} for all j.

$$Q = \sum_i (f_{ii} - a_i^2). \quad (4)$$

In the MATLAB environment, Modularity can be computed on GPU by using the following code, in which *adjMat* is adjacent matrix, and *commDiv* stores community information of all vertices. Both of them are defined in GPU memory.

```
% get all unique community name
>> comm = unique(commDiv);
% for all communities, do the i-loop
>> for i = comm
% for all communities, do the j-loop, GPU optimized
>>    gfor j = comm
% get vertices of the ith comm
>>       idxi = find(commDiv==i);
% get vertices of the jth comm
>>       idxj = find(commDiv==j);
% get edges connect ith and jth comm
>>       adjtmp = adjMat(idxi,indexj);
% the num of edges connect ith and jth comm
>>      Fij(i,j) = nnz(adjtmp);
% if f_ii computed, half the result
>>    if(i==j), Fij(i,j) = Fij(i,j)/2, end
>>    gend
>>    A(i) = sum(Fij(i,:)); % a_i in expression
>> end
% normalize matrix Fij and vector A
>> Fij = Fij/sum(A);  A= sum(A);
>> Q=trace(Eij)-sum(A.^2); % Modularity
```

4.2.7. Degree distribution

The degree distribution $P(k)$ expresses the fraction of vertices in a network with degree k.[15]

In the MATLAB environment, degree distribution can be computed on GPU by using the following code:

```
% transfer CPU matrix to GPU memory
>> gadjMatrix = gdouble(adjMatrix);
% get degree of all vertices
>> Degree = sum(gadjMatrix);
% using hist function to get the distribution
>> distribution = hist(Degree, 1000);
```

4.3. Experimental results

To compare the performance of GPU and CPU when network algorithms are given, seven benchmarks introduced in previous subsection are implemented on both CPU and GPU, and configuration of the computer is introduced in a previous subsection too. Figure 6 is the speedup bar of those benchmarks when 10000 vertices and 80000 edges are given. Because of the limits of memory, block technique is used. The tick on the left margin gives the names of benchmarks, which are CD (count degree), RN (generate random networks), APL, CC (clustering coefficient), DC (degree correlations), MOD (modularity), and DD (degree distribution) respectively. The length of the bar gives the speedup of GPU. The average speedup is 11.3, and 16.2 for maximal. However, when a smaller network is used (only containing 500 vertices and 2000 edges), the speedup is only 4.40 for average as shown in Fig. 7. If the network scale diminishes further (containing 100 vertices and 400 edges), the performance of GPU M-code drops further, sometimes even worse than its CPU counterpart. This is mainly caused by the big start-up cost of GPU computing, the data-transfer cost between CPU and GPU, and the comparatively trivial performance enhancement obtained by GPU.

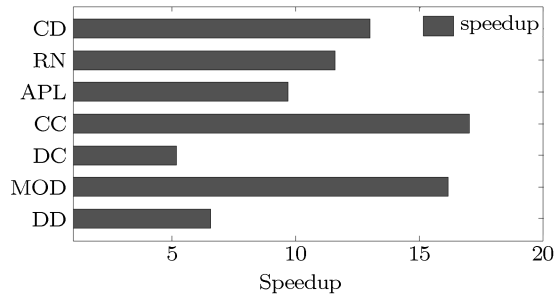


Fig. 6. Speedup of GPU M-code for networks of 10000 vertices.

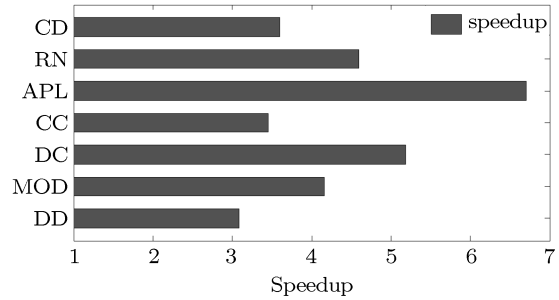


Fig. 7. Speedup of GPU M-code for networks of 500 vertices.

5. Related work

There was a great deal of research into features of complex networks,[16−18] but little attention was paid to the speed of algorithms. There was also a great deal of work about utilizing GPU resource more efficiently,[19−21] but all of this was based on the CUDA platform. Some work using GPU in high level language also existed.[22] The Jacket Toolbox developed by the AccelerEyes[10] corporation, the GPUmat Toolbox developed by the GP-you group,[23] and the MATLAB Parallel Computing Toolbox were three GPU Toolboxes of MATLAB.[24] MATLAB GPU Toolboxes were used in image processing,[25] biology applications,[26] and other areas.[27−29] However, to the best of our knowledge, there have not been any studies on complex networks based on the GPU platform and the MATLAB interface.

6. Conclusion

By leveraging the power of GPU within the MATLAB environment, great enhancement to the performance can be achieved. The advantages of the method presented in this paper are as follows.

1) Easy and rapid prototyping. For our experiment, the length of M-code is only about 1/7 as long as its C language counterpart.

2) High performance cost. Not super-computer, just a GPU added, great improvement in performance can be obtained.

3) Accelerating the execution. Speedup of GPU code archives 11.3 from the experiment in comparison with the CPU counterpart.

In a word, the combination of GPU hardware and MATLAB software with Jacket Toolbox enables high-performance solutions in normal server. Engineers and scientists of complex networks will benefit greatly from it. While this is a great success, there is still much more work which needs to be done in the future before real advantages can be seen from using a GPU to speed up general complex networks applications.

References

[1] Watts D J and Strogatz S H 1998 *Nature* **393** 440
[2] Barab A L and Albert R 1999 *Science* **286** 509
[3] Newman M E and Girvan M 2004 *Phys. Rev.* E **69** 26113
[4] Barrat A, Barthelemy M and Vespignani A 2004 *Phys. Rev. Lett.* **22** 228701

[5] Albert R and Barab A L 2002 *Rev. Mod. Phys.* **74** 47
[6] Newman M E 2003 *SIAM Rev.* **45** 167
[7] Cui D, Gao Z Y and Zhao X M 2008 *Chin. Phys.* B **17** 1703
[8] Zhang C X, Li H and Lin P 2008 *Chin. Phys.* B **17** 4458
[9] NVIDIA 2003 *SIAM Review* **45** 167
[10] The AccelerEyes 2011 *Jacket User Guide* http://www.accelereyes.com
[11] Nickolls J 2010 *Stanford CS 193G Lecture 15* **56** 72
[12] Hennessy J L, Patterson D A and Goldberg D 2003 *Computer Architecture: A Quantitative Approach* 3rd edn. (San Francisco: Morgan Kaufmann) pp. 128–132
[13] Newman M 2002 *Phys. Rev. Lett.* **89** 208701
[14] Pei W D, Chen Z Q and Yuan Z Z 2008 *Chin. Phys.* B **17** 373
[15] Rodrigues F, Travieso G and Boas V 2007 *Adv. Phys.* **56** 167
[16] Shen Y 2011 *Chin. Phys.* B **20** 511
[17] Tang Y, Wong W K, Fang J A and Miao Q Y 2011 *Chin. Phys.* B **20** 513
[18] Wang S J, Wu Z X, Dong H R and Chen G R 2011 *Chin. Phys.* B **20** 903
[19] Ryoo S, Rodrigues C I, Stone S S, Stratton J A, Ueng S Z, Baghsorkhi S S and Hwu W W 2008 *Journal of Parallel and Distributed Computing* **68** 1389
[20] Ryoo S, Rodrigues C I, Stone S S, Baghsorkhi S S, Ueng S Z, Stratton J A and Hwu W W 2008 *Program Optimization Space Pruning for a Multithreaded GPU*
[21] Shane R, Ryoo S, Rodrigues C I, Baghsorkhi S S, Stone S S, Kirk D B, *et al.* 2007 *Parallel Comput.* **33** 663
[22] Messmer P, Mullowney P J and Granger B E 2008 *Computing in Science and Engineering* **10** 70
[23] The GP-you group 2011 *GPUmat User Guide*
[24] Choy R and Edelman A 2005 *Proc. IEEE* **93** 331
[25] Kong J, Dimitrov M, Yang Y, Liyanage J, Cao L, Staples J, Mantor M and Zhou H 2010 *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* pp. 78–85
[26] Szafaryn L G, Skadron K and Saucerman J J 2009 *Biomedicine in Computing: Systems, Architectures, and Circuits* **32** 1
[27] Moore N and Leeser M 2009 *High Performance Embedded Computing Workshop*
[28] Brodtkorb A R 2008 *Scientific Computing on Heterogeneous Architectures* **12** 109
[29] Fatica M and Jeong W K 2007 *Proceedings of the Eleventh Annual High Performance Embedded Computing Workshop*, Lexington, Massachusetts 2007 pp. 18–20