2010 16th International Conference on Parallel and Distributed Systems

# Making Human Connectome Faster: GPU Acceleration of Brain Network Analysis

Di Wu*, Tianji Wu*, Yi Shan*, Yu Wang*, Yong He†, Ningyi Xu‡ and Huazhong Yang*
*Department of Electronic Engineering,
Tsinghua National Laboratory for Information Science and Technology, Tsinghua University
Email: {wud07,wutj06,shany08}@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn
† State Key Laboratory of Cognitive Neuroscience and Learning, Beijing Normal University
‡Hardware Computing Group, Microsoft Research Asia

***Abstract*—The research on complex Brain Networks plays a vital role in understanding the connectivity patterns of the human brain and disease-related alterations. Recent studies have suggested a noninvasive way to model and analyze human brain networks by using multi-modal imaging and graph theoretical approaches. Both the construction and analysis of the Brain Networks require tremendous computation. As a result, most current studies of the Brain Networks are focused on a coarse scale based on Brain Regions. Networks on this scale usually consist around 100 nodes. The more accurate and meticulous voxel-base Brain Networks, on the other hand, may consist 20K to 100K nodes. In response to the difficulties of analyzing large-scale networks, we propose an acceleration framework for voxel-base Brain Network Analysis based on Graphics Processing Unit (GPU). Our GPU implementations of Brain Network construction and modularity achieve 24x and 80x speedup respectively, compared with single-core CPU. Our work makes the processing time affordable to analyze multiple large-scale Brain Networks.**

***Keywords*-GPU; hardware computing; Human Connectome; Voxel based Brain Network**

## I. Introduction

Recently, the descriptions of structural and functional connectivity of the human brain (i.e., human connectome) have attracted considerable attention[1]. These studies are important for understanding the structure and function of the human brain in health and diseases.

The human brain is structurally and functionally organized into complex networks allowing the segregation and integration of information processing. Recent studies have suggested that a combination of multi-modal brain magnetic resonance imaging (MRI) techniques (e.g., structural MRI, functional MRI and diffusion MRI) together with graph theory approaches can help us to noninvasively map structural and functional connectivity patterns of the human brain. These approaches are particularly crucial in both neuroscience and clinics since (i) they provide insights into the understanding of the organizational principles of large-scale Brain Networks that underlie high-level cognition, and (ii) they could offer novel routes to elucidate the biological mechanisms of brain diseases and further help us to uncover network-based biomarkers for the diagnosis and monitoring of diseases[2], [3], [1].

This work is supported by Microsoft Research Asia and AMD China University Program. This work is also partially supported by National Natural Science Foundation of China (No.60870001), 863 project (No. 2009AA01Z130).

For instance, a recent functional MRI study demonstrates that the topological parameters of brain functional networks can discriminate early Alzheimer's disease patients from healthy elders with a high sensitivity of 72% and specificity of 78% [4]. Although the network-based research strategy is impressive, the detailed connectivity patterns in Alzheimer's disease still remain unclear since the whole brain is represented by only 90 nodes (regions) in the previous study. A comprehensive, detailed analysis by including thousands of network nodes derived from neuroimaging *voxels* is important and necessary in the Brain Network research.

From the existing, non-invasive imaging techniques, the brain can be represented by 20k to 100k voxels. A fine-scale voxel based Brain Network can be constructed by measuring the structural or functional relationship between all pairs of image voxels. However, both the construction and analysis of voxel level Brain Networks require tremendous computation power. A coarse-scale region based network is built by first average the acquired data within each brain region, then regard each region as a network node. There is a big loss of information in the building of region based networks.

Nowadays, much work has been done to construct and analyze the Brain Networks, but most of them was focused on the coarse-scale region level networks. For instance, several studies have utilized a prior brain atlas to parcellate the brain into tens of brain regions and then constructed region-based Brain Networks[5], [6]. Other studies have used image voxels to build a partial Brain Network at a fine scale[7]. Recently, there is some work on voxel based Brain Network analysis[8]. However, some approximate algorithms (such as random walk method) were used to avoid complex eigenvectors computation of the correlation matrix.

According to the above, the Brain Network research is a potential customer of high performance/low power hardware computing. Figure 1 illustrates the importance of high-performance/low-power hardware computing in the Brain Network research.

The computation strategy can be divided into two categories. In the first one, small-scale computing nodes such as personal computers are used in hospitals, since they are convenient and cheap for maintenance. These computers can be configured with dedicated acceleration hardware (such as GPUs and FPGAs) to fit the specific problem domain. In the second

1521-9097/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPADS.2010.105

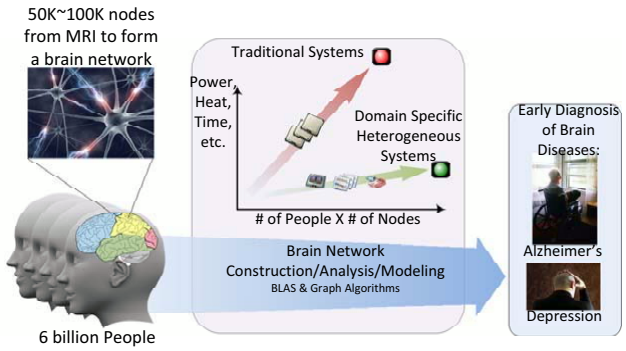



Fig. 1. To analyze and model Brain Networks for early diagnosis of brain diseases, we need to sample as many people as possible, with the resolution as high as possible. However, when the number of people and the number of Brain Network nodes increase, traditional computing systems will take either unaffordable time (perhaps years) or unaffordable electricity power (perhaps mega-watts). With efficient hardware computing systems, both the speed and the power will be reasonable (perhaps days/ hundreds of watts).

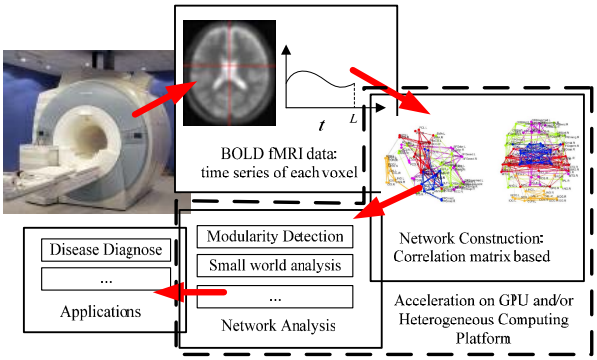



Fig. 2. Brain Network Analysis Framework

category, large-scale high-performance computers are used in research institutes. Those computers with dedicated hardware are grouped into clusters to form a heterogeneous hardware computing platform. Its high efficiency and computation speed are especially crucial for the analysis of the Brain Networks with huge sizes and a very large sample capacity.

Among the hardware computing platforms, general purpose GPU emerges as a very powerful and low-cost parallel computing platform. GPGPU has been used in many applications, such as linear algebra, graphic or network based algorithms. In [9], a framework for linear algebra operators on GPU is proposed. In [10], several fundamental graph algorithms are implemented on GPU, such as breadth first search, single source shortest path, and all-pairs shortest path. In [11], a method is proposed for obtaining the all-pairs shortest path for large graphs on GPUs. However, few of them are designed for the Brain Network Analysis.

In this paper, we, for the first time, propose a GPU based brain network analysis framework to accelerate the analysis of large scale Brain Networks. Under this framework, we accelerate the construction and modularity of Brain Networks by 24x and 80x using AMD GPU platform. Our voxel based Brain Networks consist of 38368 nodes.

The rest of this paper is organized as follows. Section II introduces our GPU based Brain Network analysis platform and the key algorithms, including construction and modularity operations. Section III proposes the GPU implementation of these two algorithms. Experimental results are shown and analyzed in Section IV. Section V concludes the paper and puts forward the future work.

## II. Framework and Algorithms

In this section, we introduce our GPU framework for Brain Network construction and analysis, and the algorithms used in this work.

The original data is acquired from functional MRI, which provides the blood oxygen level dependent (BOLD) signal of each voxel at a certain spatial resolution. By sampling the signal at a certain frequency for a period of time, we get a time series of BOLD signal for each voxel. Hence, the data acquired from fMRI is represented by a 4-dimension matrix: the x-, y- and z-axis denote to the spatial position of voxels in the human brain and the t-axis represents the BOLD sequence of a voxel. In our experiments, the grey matter contains $N_v = 38368$ voxels, and the BOLD signal of each voxel is $L = 230$ points in length.

A voxel-based Brain Network can be built from the BOLD sequences of all voxels. It is a network that illustrates the connections between these voxels. Each node in the network represents a voxel, and each connection represents the correlation between the BOLD signals of the pair of voxels. After the Brain Network is built, graph algorithms can be applied to analyze the network, such as network hub detection, modularity detection, small-world analysis, etc. Our framework is illustrated in Figure 2.

In this work, we focus on the construction of Brain Networks and one aspect of the network analysis, i.e. modularity detection of the network.

### A. Brain Networks Construction

Here we introduce the detailed algorithm of brain-network construction. This algorithm is used in [5], [8].

By fMRI, a series of signal of length $L$ is acquired for each of the $N_v$ voxels. For each pair of nodes (voxels) $(v_i, v_j)$, we obtain the Pearson's correlation [12] between the series of the pair, i.e.

$$\hat{r}_{i,j} = \frac{\sum (v_i - \bar{v_i})(v_j - \bar{v_j})}{\sqrt{\sum (v_i - \bar{v_i})^2 \sum (v_j - \bar{v_j})^2}} \quad (1)$$

$$= \frac{\sum v_i v_j - \frac{1}{n}(\sum v_i)(\sum v_j)}{\sqrt{\left(\sum v_i^2 - \frac{1}{n}(\sum v_i)^2\right)\left(\sum v_j^2 - \frac{1}{n}(\sum v_j)^2\right)}} \quad (2)$$

where $v_i$ denotes to the series of voxel i, $\bar{v_i}$ is the average of the series of that voxel, and all $\sum$ denotes to $\sum_{t=0}^{L-1}$, i.e. summing along the whole time series.

From equation 2, we can see that only the term $\sum_{t=0}^{L-1} v_i v_j$ needs to be calculated for each pair of voxels, while the first

moment $\sum v_i$ and second moment $\sum v_i^2$ can be calculated on a per-voxel basis.

The absolute magnitude of correlation, i.e. $r_{i,j} = |\hat{r_{i,j}}|$ of a pair of voxels represents the strength of connection of the pair. After calculating the magnitude of correlation of all pairs, we get a symmetric correlation matrix. The matrix represents a complete graph of all voxels with weighted edges. From the matrix, we can quickly obtain the weight of each voxel by summing up the weight of all edges connected with the voxel, i.e.

$$w_i = \sum_{j \neq i} r_{i,j} \tag{3}$$

Voxels with higher weight tend to be located in the hub regions of the brain.

In this work, we focus on un-weighted Brain Networks, which can be built from the correlation matrices by applying a threshold on the weight of edges. Edges that are weighted higher than the threshold remain to be connections in the un-weighted network, while other edges are discarded. The threshold should be chosen to ensure the un-weighted network is connected and has a certain sparsity $S$. $S$ is defined to be the ratio of existing edges in a network to the number of total possible edges. The un-weighted network can be represented by a sparse matrix, in which all non-zero numbers are ones, or by an adjacency list.

Un-weighted networks are less memory-consuming compared to weighted networks. In our experiment, the full correlation matrix has the size of $38368 \times 38368$, and consumes about 2.7GB of memory when elements are in 32-bit float type and only half of the matrix is stored due to its symmetry. It takes much less space to store a sparse matrix depending on the sparsity.

Although the threshold can be applied on the fly when calculating the correlation matrix, the amount of calculation for obtaining a full correlation matrix can not be released. There are $N_{pair} = N_v(N_v - 1)/2$ pairs. By calculating the order 1 and 2 moments of all voxels in advance, obtaining the correlation of each pair requires roughly $2L$ floating-point operations (FLOP). Hence, the construction of a Brain Network has $O(N_v^2 L)$ complexity.

### *B. Brain Networks Modularity*

After constructing the voxel-base Brain Network, we examine the methods to analyze the modular organization of it using graphic modularity algorithms. There are several methods that are applicable to un-weighted sparse adjacent networks. A random-walk-based method is introduced in [13], and is used in [8]. A greedy algorithm is presented in [14], and is used in [5].

The algorithm we choose for Brain Network community structure detection is the eigenvector-based spectral partition method[15]. The idea of modularity is to find groups of points that has a lot of inner-group connections and few inter-group connections. A benefit function $Q$ is introduced to judge the network's modularity, which is defined as follow:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - P_{ij}\right] \delta\left(g_i, g_j\right) \tag{4}$$

where $A_{ij}$ is the binary adjacent matrix representing the Brain Network; $P_{ij}$ is the probability for an edge to fall between every pair of vertices $i$, $j$; $g_i$ is defined as the community to which vertex $i$ belongs; $\delta\left(g_i, g_j\right)$ is 1 if $g_i = g_j$ and 0 if otherwise, and $m$ is the number of edges in the network. $P_{ij}$ can be defined as $P_{ij} = \frac{k_i k_j}{2m}$,where $k_i$ is the degree of node $i$. When dividing the network into only two groups, we let $s_i$ be the indicator of the division: $s_i = 1$ if the node $i$ belongs to one group and -1 if it belongs to another. Then the modularity can be denoted as:

$$Q = \frac{1}{4m} \sum_{ij} \left[A_{ij} - P_{ij}\right](s_i s_j + 1) \tag{5}$$

$$= \frac{1}{4m} \sum_{ij} \left[A_{ij} - P_{ij}\right] s_i s_j \tag{6}$$

The latter derivation is based on the fact that $\sum_{ij} A_{ij} = \sum_{ij} P_{ij}$. Then Q can be rewritten to a matrix form by defining $\mathbf{B}$:

$$B_{ij} = A_{ij} - P_{ij} \tag{7}$$

a real symmetric matrix, called *Modularity Matrix*.

Then the problem becomes finding the best division $\mathbf{s}$ that maximize $Q$. In [15], it can be proven that the best $\mathbf{s}$ can be obtained by the eigenvector $\mathbf{u}$ of $\mathbf{B}$ with the most positive eigenvalue, i.e.

$$s_i = \begin{cases} 1, & \text{if } u_i \geq 0, \\ -1, & \text{if } u_i < 0 \end{cases} \tag{8}$$

Hence, using the eigenvector corresponding to the most positive eigenvalue of $\mathbf{B}$, we can divide the network into two groups according to the signs of the elements of this eigenvector.

The Brain Networks are unlikely to have only two communities. A modified algorithm to handle multiple division is also described in [15]. Firstly, the benefit function $Q$ is modified to $\Delta Q$, which is the increment of $Q$ before and after subdivision of the community, thus:

$$\Delta Q = \sum_{i,j \in G} \sum_{k=1}^{c} B_{ij} s_{ik} s_{jk} - \sum_{i,j \in G} B_{ij} \tag{9}$$

$$= \sum_{k=1}^{c} \sum_{i,j \in G} \left[ B_{ij} - \delta_{ij} \sum_{l \in G} B_{il} \right] s_{ik} s_{jk} \tag{10}$$

$$= \mathbf{Tr}[\mathbf{s}^T \mathbf{B}^{(G)} \mathbf{s}] \tag{11}$$

where $\mathbf{B}^{(G)}$ has the elements:

$$B_{ij}^{(G)} = B_{ij} - \delta_{ij} \sum_{l \in G} B_{il} \tag{12}$$

Similarly, the division can be found according to the signs of the elements of the eigenvector $\mathbf{u}$ corresponding to the most positive eigenvalue $\beta$ of $\mathbf{B}^{(G)}$. The division maximizes $\Delta Q$, thus maximizes the contribution to the increase of $Q$ of the whole network. The algorithm stops when there is no positive eigenvalue, which means there is no division that can increase the modularity of the network.

## III. GPU Implementations

In this section, the architecture of AMD GPU platform is introduced first, followed by the description about the GPU accelerated implementations of the construction and modularity analysis of Brain Networks.

### A. AMD GPUs

We choose the general purpose GPU platform from AMD, named ATI Stream. In this section, we briefly discuss the hardware functionality and programming model of AMD RV870 GPUs, which belong to the latest family of AMD GPUs. Please refer to [16] for detailed information.

*1) Computation Units and Programming model:* In RV870, stream cores or ALUs are organized as 5-way VLIW processors, called thread processors. Each thread processor contains 4 normal cores that can perform 32-bit integer or floating-point arithmetic, and 1 transcendental core that can perform transcendental functions such as trigonometric or exponential functions.

16 thread processors are grouped into a SIMD engine. All thread processors in a SIMD engine performs the same instruction at any time, but on there private registers; different SIMD engines can perform different instructions. In a programmer's view, the width of each SIMD engine is 64 due to hardware switching of threads. The bundle of 64 threads that simultaneously run on a SIMD is called a wavefront. The run time of a wavefront is determined by the slowest thread in it. In RV870, there are 20 SIMD engines.

There are two types of kernels (programs run on the GPU), pixel shader (PS) and compute shader (CS). Here we only introduce the CS, in which threads are organized in groups. Each group consists of 1 or more wavefronts. These wavefronts are guaranteed to be run on the same SIMD engine, and thus can share data through the local data share memory (LDS, introduced later). Threads in different group can not share data through the LDS.

*2) Memory Hierarchy:* In RV870, several memory resources can be used, each with different accessing constraints and speed.

*general purpose registers (GPRs)* are the fastest memories. Each thread has access to up to 127 GPRs in float4 type, which is a short vector with 4 single precision floating point elements, named x, y, z and w.

*local data sharing memory (LDS)* - Each SIMD engine has a 32KB dedicated LDS memory which enables low latency data sharing between threads in the same SIMD. On RV870, the LDS is organized in $32-bank \times 256-row$ structure. Each memory entry is 32-bit wide. Multiple threads accessing to a same bank will result in bank conflict, and the access will be serialized. The LDS supports random access and several atomic operations.

*off-chip graphic memory* is the largest and slowest memory resource. It supports several access models: image, UAV or global buffer. In CS mode, we regard input-only resources as images, since image resource supports *Texture sampling*, which is cached reading from the memory to GPRs. Linear buffers can be regarded as *Uniform Access Views (UAVs)* or the *Global buffer*, which have read/write access for all threads. On RV870, several UAVs are supported, however, only one global buffer is supported. UAVs and the global buffer also support several atomic operations.

### B. Brain Networks Construction

As discussed in Section II-A, to construct a functional Brain Network from fMRI data, we need to obtain the Pearson's correlation of all pairs of voxels. This operation is very suitable for mass parallel processors such as GPUs, since the computation of different pairs of voxels can be fully parallelized.

Before the correlation matrix calculation, the first and second moments, i.e. $\sum_{t=0}^{L-1} v_i(t)$ and $\sum_{t=0}^{L-1} v_i^2(t)$ of all series are obtained. The moments and series of all voxels are transferred to the graphic memory. The space requirements are in the order of $O(N_v L)$. The GPU kernel computes the cross correlation term of each pair of voxels, i.e. $\sum_{t=0}^{L-1} v_i(t)v_j(t)$ for each pair $(i, j)$.

A straightforward way of implementing the construction algorithm is to use each of GPU thread to calculate one correlation. However, there are several characters and constraints of the GPU platform that should be taken into consideration.

**Algorithm 1:** Construction of the Correlation Matrix

**input** : Series of voxels,
$v_i(t), i = 0..N_v - 1, t = 0..L - 1$
First and second moments of series,
$m_i^{(1)}, m_i^{(2)}, i = 0..N_v - 1$
**begin**
  Get corresponding $i$ and $j$ from Thread/Group ID
  $t \leftarrow ThreadID$; $acc \leftarrow 0$;
  **while** $t < L$ **do**
    $acc \leftarrow acc + v_i(t) * v_j(t)$;
    $t \leftarrow t + 64$;
  **end**
  Binary reduction of $acc$ in LDS, the result is stored in $acc$ of Thread 63
  **if** *ThreadID==63* **then**
    $r_{i,j} = |\frac{acc - m_i^{(1)} m_j^{(1)}/n}{\sqrt{\left(m_i^{(2)} - m_i^{(1)2}/n\right)\left(m_j^{(2)} - m_j^{(1)2}/n\right)}}|$;
    Output $r_{i,j}$;
  **end**
**end**

In our implementation, we use 64 threads (i.e. a wavefront) to calculate each correlation, where 64 is the width of SIMD engine in AMD GPUs we use. The pseudo-code of GPU kernel is presented in Algorithm 1. Threads in a wavefront fetch the corresponding data and calculate $\sum_{k=0}^{(L-1)/64} v_i(64k + Tid)v_j(64k + Tid)$ (where $Tid$ is the ID of threads within a wavefront) before joining there intermediate results into one, using binary reduction, and calculate $r_{i,j}$. The series of data of each voxel is consecutively stored in graphic memory. Hence, by using a wavefront to obtain one correlation, memory fetch can be coalesced, illustrated in Figure 3.

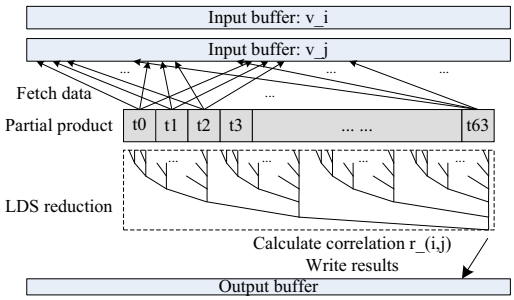



Fig. 3. Network Construction: a wavefront coalesced fetches the data of $v_i$ and $v_j$; each thread calculates a partial result of the dot product and performs a wavefront-wide reduction in the LDS; thread T63 finally computes the correlation and writes to the output buffer.

The full correlation matrix, with $O(N_v^2)$ space requirement, is too large to fit into the graphic memory. Considering its symmetry, only the upper half of the matrix is calculated. However, it is still not applicable to invoke the GPU kernel once and keep the results on the graphic memory before the kernel ends. In our implementation, a fixed amount of data is calculated for each kernel invocation. This piece of data is transferred back on to the system memory while the GPU kernel is invoked again for another tile of data. CPU will post-process the tile of resultant data if we expect the binary matrix represented as adjacency list. This implementation can be easily scaled to build networks with higher resolution, without the demand for higher graphic memory space.

### C. Brain Networks Modularity

According to Section II-B, the community structure detection requires huge amount of computation for eigenvalues and eigenvectors, which makes this approach impractical when the network scale becomes extremely large. However, compared to the variety of fast algorithm of network modularity in the past, this approach is particularly effective in producing good results[15].

The core operation of the modularity detection algorithm (See Section II-B) is the computation of $\mathbf{u}$ and $\beta$, which is done for each subdivision. First, we use the power method[17] to compute the leading eigenvalue $\hat{\beta}$. If $\hat{\beta} \geq 0$, $\beta = \hat{\beta}$. Otherwise, when $\hat{\beta} < 0$, we shift the matrix as $\mathbf{B}^{(G)} - \hat{\beta}\mathbf{I}$, so that all the eigenvalues of the new matrix are nonnegative. Hence, $\beta = \beta_1 - \hat{\beta}$ is the needed eigenvalue of the original matrix, where $\beta_1$ is the leading eigenvalue of the shifted matrix, also obtained by the power method.

By examining equation 7 and 12 more carefully, it could be noticed that:

$$\sum_{l \in G} B_{il} = \sum_{l} A_{il} - \frac{k_i k_l}{2m} = \sum_{l} A_{il} - \frac{k_i}{2m} \sum_{l} k_l \qquad (13)$$

The two sums would not change throughout the entire power method, so they could be computed only in the beginning of each round. The sum $\mathbf{d}$ is defined as follows:

$$d_i = \sum_{l} A_{il} - \frac{k_i}{2m} \sum_{l} k_l \qquad (14)$$

To compute $\mathbf{d}$, we first define $\mathbf{s^{(k)}}$, the $k$th separator:

$$s_i^{(k)} = \begin{cases} 1, & \text{if } u_i^{(k)} \geq 0, \\ 0, & \text{if } u_i^{(k)} < 0 \end{cases} \qquad (15)$$

Then $\mathbf{d}$ can be compute as follow:

$$\mathbf{d}^{(i)} = \mathbf{A} \times \mathbf{s}^{(i)} - \frac{\mathbf{k}^T \times \mathbf{s}^{(i)}}{2m} \mathbf{k} \qquad (16)$$

The fact that the dimension of $\mathbf{B}^{(G)}$ changes in subdivisions (See Equation 12), rendering the eigenvalue and eigenvector computation more complicated. To solve this problem ,$\mathbf{B'}$, an $N_v$-dimensional matrix is introduced:

$$\mathbf{B'} = \mathbf{B} - diag\{d_1, \ldots, d_n\} \qquad (17)$$

$$= \mathbf{A} - \frac{\mathbf{k}^T \times \mathbf{k}}{2m} - diag\{d_1, \ldots, d_n\} - \beta \mathbf{I} \qquad (18)$$

In each iteration of power method:

$$\mathbf{Y}_k = \frac{\mathbf{B'} \times \mathbf{Y}_{k-1}}{\|\mathbf{B'} \times \mathbf{Y}_{k-1}\|_2} \qquad (19)$$

$$= \mathbf{A}\mathbf{Y}_{k-1} - \frac{\mathbf{k}^T \mathbf{Y}_{k-1}}{2m} \mathbf{k} - diag\{d_1, \ldots, d_n\}\mathbf{Y}_{k-1} - \beta \mathbf{Y}_{k-1} \qquad (20)$$

$\mathbf{Y}_k$ is the vector of the $k$th iteration, $\beta$ is the leading eigenvalue if it is negative and 0 in the beginning. We notice that we could multiply $\mathbf{Y}_{k-1}$ with $\mathbf{s^{(i)}}$:

$$y_j^{'(k-1)} = y_j^{(k-1)} * s_j^{(i)},\ j = 1 \ldots n \qquad (21)$$

Then we multiply the result $\mathbf{Y}_k$ with $\mathbf{s^{(i)}}$. In this way, the dimension of $\mathbf{B'}$ remains unchanged during the power iterations.

To sum up, we re-organize Newman's algorithm to a parallel program. We specifically design the implementation based on AMD GPU architecture by partitioning the serial algorithm into several basic linear algebra operations. The system chart of our modularity implementation is illustrated in Figure III-C.

The modularity detection algorithm is based on iterations. Each iteration of the outer loop generates a subdivision vector based on the eigenvector. There is also an inner loop inherent in the power method for eigenvector calculation. Hence, the algorithm is fundamentally serial. As a result, it is difficult to migrate the entire algorithm to GPU platform. Instead, we implement a finer granularity acceleration of the algorithm. That is, we designed the basic matrix operations such as CSR sparse matrix multiplication and vector addition and multiplication. The general scheduling is performed by CPU and the most time-consuming matrix calculations is done by GPU. As the vectors and sparse matrix is loaded to GPU buffer before the calculation and remain unchanged throughout the computation, our approach fully utilized the GPU's advantages in parallel computing.

*1) Eigenvalue calculation:* The most resource consuming part of the algorithm is the computation of leading eigenvalues, which is generally difficult when the matrix are large. An effective method is the power method[17], in which an approximate eigenvector is iteratively multiplied by the matrix, until convergence. This method is particularly effective for large sparse matrices.

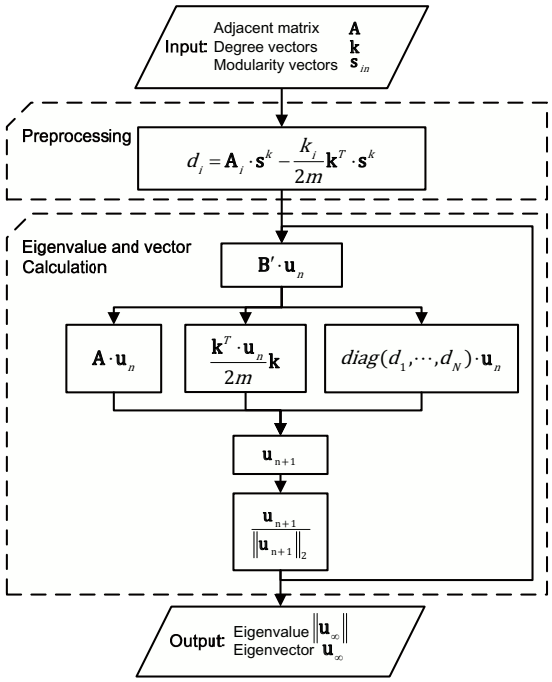



Fig. 4. System Chart of our GPU implementation of Newman's Modularity Algorithm.

In our case, although our Brain Network is represented by sparse real-symmetric matrix, the actual matrix that need eigenvalue computing is $\mathbf{B^{(G)}}$ or $\mathbf{B}^{(G)}$, as we defined at Section II-B, which is dense. Dense matrix-vector multiplication is a time and space consuming operation. Fortunately, we find that the computation can be divided into two parts: one of them is multiplication between sparse matrix and vector, the other is multiplication between vectors. The pseudo-code is shown in Algorithm 2.

*2) Partition:* In the previous Section II-B, finding the division of a network is to find $\mathbf{s}$ which is parallel to the eigenvector of equation 7 corresponding to the most positive eigenvalue. Then we defined the separators $\mathbf{s^{(i)}}$ who simplified out calculation. The final output of our modularity program is a $N_v \times M$ vector group $\mathbf{S}$, $N_v$ is the number of vertices in the network and $M$ is the number of communities we detected. We have

$$S_{ij} = \begin{cases} 1, & \text{if vertices } j \text{ belongs to community } i, \\ 0, & \text{otherwise} \end{cases} \quad (22)$$

After the eigenvector computation, the ones in $\mathbf{s}^{(i)}$ are divided into two groups according to the signs of each elements of the eigenvector. So the whole process of division is performed as a top-down tree structure, each branches will continue to divide until the biggest eigenvalue becomes negative or less than a threshold, which we can use to control the number of partitions. We illustrate the partition flow in Figure III-C2.

**Algorithm 2:** Eigenvalue computation of Modularity

**input** : CSR sparse matrix $\mathbf{A}$,
a $N_v \times 1$ vector $\mathbf{s}_{in}$,
a $N_v \times 1$ vector $\mathbf{k}$
**output**: float point eigenvalue $\beta$,
a $N_v \times 1$ vector $\mathbf{s}_{out}$
**begin**
  $\beta_{bias} \leftarrow 0$;
  $\mathbf{d} = \mathbf{A} \times \mathbf{s}_{in} + \frac{\mathbf{k}^T \times \mathbf{s}_{in}}{2m}\mathbf{k}$;
  **while** $signal < 0$ **do**
    *initialize* $\mathbf{x}$; $\mathbf{x} \leftarrow \mathbf{x}. * \mathbf{s}_{in}$;
    $degree \leftarrow \mathbf{k}^T \times \mathbf{x}$;
    **while** $|\beta - \beta^{'}| < \epsilon$ **do**
      $\mathbf{x}^{'} \leftarrow \mathbf{x}$; $\beta^{'} = \beta$;
      $\mathbf{y} \leftarrow \mathbf{A} \times \mathbf{x}$;
      $\mathbf{y} \leftarrow \mathbf{y} - degree \cdot \mathbf{k} - \mathbf{d}. * \mathbf{x} - \beta_{bias}\mathbf{x}$;
      $\mathbf{y} \leftarrow \mathbf{y}. * \mathbf{s}_{in}$;
      $\beta \leftarrow \|\mathbf{y}\|_2$;
      $\mathbf{x} \leftarrow \mathbf{y}$;
    **end**
    **if** $max|x_i| \cdot max|x_i^{'}| > 0$ **then** $signal \leftarrow 1$;
    **else** $signal \leftarrow -1$;
    $\beta_{bias} \leftarrow signal \cdot \beta$;
  **end**
  **for** $i \leftarrow 0$ **to** $N_v - 1$ **do**
    **if** $x_i > 0$ **then** $s_i^{(out)} \leftarrow 1$;
    **else** $s_i^{(out)} \leftarrow 0$;
  **end**
**end**

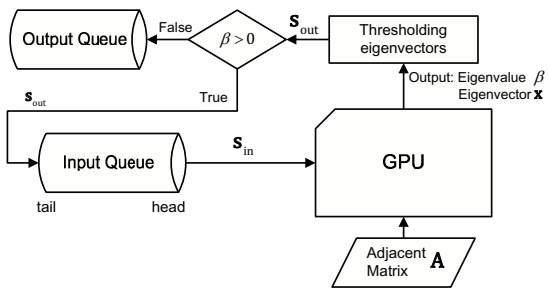



Fig. 5. Data input & output flow chart of our modularity implementation.

## IV. Experimental Results

The computing platform in our experiments has a quad-core Phenom II 956 CPU running at 3.4GHz, 8GB DDR3 memory, a Radeon HD 5870 graphic card with RV870 core at 850MHz and 1GB GDDR5 memory. Our GPU kernels are written in ATI Intermediate Language (IL) [16].

### A. Construction

The data we use in our experiments are acquired from BOLD fMRI scanning. For now, we only handle the grey

matter of human brain, which contains $N_v = 38368$ voxels at 3mm resolution.

Table I shows the comparison of running time for constructing Brain Networks between our GPU implementation and single-core CPU. To better analysis the performance and bottlenecks of the GPU implementation, we divide the execution time into several parts. The Moments time is the cost of 1st and 2nd moments calculation including the transfer cost of MRI data to the GPU memory. The adjacent list computing time includes correlation calculation, data transfer and threshold applying. The Correlation time is the running time of GPU kernel function, and the Transfer time is the cost of transferring correlations to CPU, as is described in Section III-B. Threshold applying is performed by CPU, so the costs are identical between single-core CPU and GPU implementations. The experimental results show that our accelerated GPU implementation achieves a 29x speedup in correlation computing and a 24x speedup overall.

TABLE I
BRAIN NETWORKS CONSTRUCTION SPEED COMPARISON (IN SECONDS)

| | Moments | Adjacent list | | | Total |
|---|---|---|---|---|---|
| | | Correlation | Transfer | Threshold | |
| GPU | 0.04 | 32.35 | 3.33 | 6.41 | 42.13 |
| CPU | 0.10 | 1020.33 | | 6.04 | 1026.47 |
| Speedup | 2.5x | 28.6x | | | 24.4x |

There are two main restraints to our GPU implementation for a better speedup. First of all, there are massive memory fetch operations compared with simple ALU operations. There are $\lceil 230/64 \rceil = 4$ read operations in each thread and one global memory write operation in each wavefront. Secondly, the parallel 'mapping' parts are too simple compared with the serial 'reduction'. As described in Section III-B, each thread performs the multiplication of at most 4 pairs of elements, but the 64th thread in the wavefront performs 6 times of LDS fetch and addition as well as the calculation of the entire Equation 2.

However, we can improve the performance of our implementation by utilizing multiple GPU platforms. Theoretically, only the correlation calculation would be influenced. Ideally, when we use 2 identical GPU platforms, the data transfer time would be doubled and the kernel function time would be halved. Based on these assumptions, we could achieve a 45x speedup in correlation calculation and a 35x speedup overall for network construction.

### *B. Modularity*

The modularity computation costs a considerable time and divides the network in to thousands of communities. The CPU version requires so much time that we cannot finish the full division of the network. In response to that, we truncate the calculation to the first 100 iterations and compare the single-core CPU results with GPU. Table II shows the speed comparison. (Note: iterations are of the outer loop of Algorithm 2.)

In practical, we prefer detecting the most significant parts of the Brain Network rather than dividing them to the smallest

TABLE II
BRAIN NETWORKS MODULARITY SPEED COMPARISON (IN SECONDS)

| | CPU | RV870 | Speedup |
|---|---|---|---|
| First 100 iterations | 74496.97 | 928.73 | 80x |
| Slowest iteration | 2515.55 | 31.68 | 79x |
| Fastest iteration | 3.78 | 0.29 | 13x |

groups. Therefore, we want to control the number of subgroups produced by the modularity algorithm. This is done by setting up a eigenvalue threshold, $r_e$. A branch of division is terminated when the eigenvalue is below the threshold rather than below 0. That is to say, we end the division when it provides not necessarily no benefits to the modularity of the entire network, but when the benefits is less than expected. However, we should point out that this approach does not suffice the optimal division under our "expectation". The reason is that our modularity method is a kind of greedy-algorithm which means each division is local optimum yet does not guarantee global optimum. This is discussed at length in [15]. Nevertheless, despite the possibility of slight inaccuracy, the general image provided by our method is acceptable. Table III shows the relationship between the eigenvalue threshold and the number of partitions of the network.

TABLE III
EIGENVALUE THRESHOLD AND DIVISIONS UNDER $r_{th} = 0.75$

| Eigenvalue Threshold ($r_e$) | Divisions | Time (sec) |
|---|---|---|
| 100 | 225 | 928.73 |
| 110 | 49 | 906.08 |
| 125 | 28 | 496.31 |
| 130 | 20 | 441.64 |
| 140 | 7 | 116.55 |
| 150 | 7 | 113.79 |
| 200 | 4 | 78.26 |

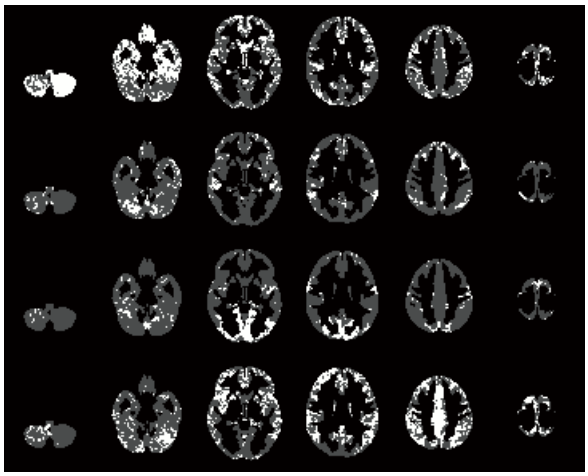
Table IV shows the profiling of our GPU implementation of Network Modularity. The results are calculated for networks constructed under $r_{th} = 0.78$ , and the number of divisions is controlled by eiganvalue threshold $r_e$. From the results we can see that the Sparse Matrix and Vector multiplication (SPMV) takes most of the time in the program. Our SPMV and basic vector operations are optimized for AMD GPU platforms. The implementation details and results of our GPU implementations of SPMV and vector operations are discussed in [18]. Furthermore, the data transfer costs little time in our implementation. The input data are the sparse adjacent matrix of the Brain Network, and the output data are the partition vector. As a result, the performance of our implementation can be further improved by multiple GPU platforms in the same way as discussed in Section IV-A.

TABLE IV
PROFILING OF THE GPU IMPLEMENTATION OF NETWORK MODULARITY UNDER $r_{th} = 0.78$

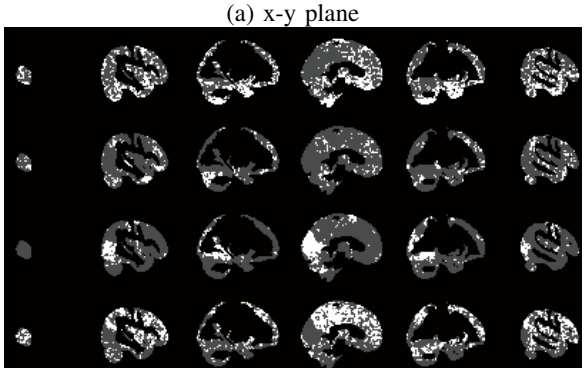
| Divisions | Data Input | SPMV | Vector OPs | Data Output | Total |
|---|---|---|---|---|---|
| 305 | 0.22 | 1734.78 | 266.33 | 0.78 | 2002.11 |
| 42 | 0.22 | 452.37 | 49.80 | 0.09 | 507.48 |
| 9 | 0.22 | 168.98 | 14.11 | 0.01 | 183.32 |

By mapping the partition results to the grey matter on the

human brain, we can display the actual position of each groups in pictures. Here we draw the images of the modularity of $r_e = 200$ and $r_{th} = 0.75$ when 4 subparts are obtained from the computation, as is illustrated in Figure 6.



(a) x-y plane



(b) y-z plane

Fig. 6. Dividing the Brain Network into 4 parts: each part corresponds to one row in the figure, marked as bright white.

## V. Conclusion

In this work, we propose a novel approach of accelerating the computation of Brain Networks' construction and analysis. We implemented the network construction and modularity analysis and achieved considerable speedups on both algorithms. Our work provides a solution against the computational limits that impede the study of voxel-based Brain Network.

In constructing the voxel-based Brain Network, we provide a higher resolution picture of the human brain. The smaller the scale of unit in the network is, the more accurate the results can be. While giving more accurate positions of the potential hubs in the network, we greatly reduce the time of voxel-based Brain Network construction.

The modularity of voxel-based Brain Network provides a clear picture of the connectivity pattern of human brain cells. In the past, most of the research on the functional characteristics of the brain has been based on the anatomical segmentation. The modularity of voxel-based Brain Network gives a new way to find the community structure of the brain based on the connectivity information of each voxels. Our acceleration of the algorithm overcomes the computation obstruction and, more importantly, produces better and more detailed results.

There is a myriad of Brain Network analytic methods that can be accelerated. We are going to form a long-term cooperation with Brain Network researchers and keep on the work of acceleration. Our implementation of Brain Network construction and modularity can be improved as well. For instance, in each division it is redundant to calculate the entire $N_v \times N_v$ matrix. The algorithm can be optimized. Moveover, our implementation can be extended to multiple GPU platforms to further improve performance.

## References


[1] O. Sporns, G. Tononi, and R. Ktter, "The human connectome: A structural description of the human brain," *PLoS Comput Biol*, vol. 1, no. 4, p. e42, 09 2005.

[2] E. Bullmore and O. Sporns, "Complex brain networks: graph theoretical analysis of structural and functional systems," *Nat Rev Neurosci*, vol. 10, pp. 186–198, 2009.

[3] Y. He, Z. Chen, G. Gong, and A. Evans, "Neuronal networks in alzheimer's disease," *Neuroscientist*, vol. 15, no. 4, pp. 333–350, 2009.

[4] K. Supekar, V. Menon, D. Rubin, M. Musen, and M. D. Greicius, "Network analysis of intrinsic functional brain connectivity in alzheimer's disease," *PLoS Comput Biol*, vol. 4, no. 6, p. e1000100, 06 2008.

[5] Y. He, J. Wang, L. Wang, Z. J. Chen, C. Yan, H. Yang, H. Tang, C. Zhu, Q. Gong, Y. Zang, and A. C. Evans, "Uncovering intrinsic modular organization of spontaneous brain activity in humans," *PLoS ONE*, vol. 4, no. 4, p. e5226, 04 2009.

[6] J. Wang, L. Wang, Y. Zang, H. Yang, H. Tang, Q. Gong, Z. Chen, C. Zhu, and Y. He, "Parcellation-dependent small-world brain functional networks: a resting-state fmri study," *Human Brain Mapping*, vol. 30, no. 5, pp. 1511–1523, 2009.

[7] D. A. Fair, A. L. Cohen, J. D. Power, N. U. F. Dosenbach, J. A. Church, F. M. Miezin, B. L. Schlaggar, and S. E. Petersen, "Functional brain networks develop from a local to distributed organization," *PLoS Comput Biol*, vol. 5, no. 5, p. e1000381, 05 2009.

[8] M. Valencia, M. A. Pastor, M. A. Fernández-Seara, J. Artieda, J. Martinerie, and M. Chavez, "Complex modular structure of large-scale brain networks," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 19, no. 2, p. 023119, 2009.

[9] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM, 2005, p. 234.

[10] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," *High Performance Computing*, vol. 4873, pp. 197–208, 2007.

[11] G. J. Katz and J. T. Kider, Jr, "All-pairs shortest-paths for large graphs on the gpu," in *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 47–55.

[12] J. L. Rodgers and W. A. Nicewander, "Thirteen ways to look at the correlation coefficient," *The American Statistician*, vol. 42, no. 1, pp. 59–66, 1988.

[13] P. Pons and M. Latapy, "Computing communities in large networks using random walks," *Journal of Graph Algorithms and Applications*, vol. 10, no. 2, pp. 191–218, 2006.

[14] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," *Phys. Rev. E*, vol. 69, no. 6, p. 066133, Jun 2004.

[15] ——, "Finding community structure in networks using the eigenvectors of matrices," *Phys. Rev. E*, vol. 74, no. 3, p. 036104, Sep 2006.

[16] *ATI Intermediate Language (IL) Specification*, Advanced Micro Devices, Inc., Dec 2009.

[17] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1999.

[18] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient pagerank and spmv computation on amd gpus," in *Proceedings of the 39th International Conference on Parallel Processing (ICPP-2010)*, 2010.