AN EFFICIENT PARALLEL IMPLEMENTATION of STRUCTURAL NETWORK CLUSTERING in MASSIVELY PARALLEL GPU

by

Recep Avci

A project report presented to the Department of Computer Science
and the Graduate School of the University of Central Arkansas
in partial fulfillment of the requirements for the degree of

Master of Science

in

Applied Computing

Conway, Arkansas

May 2013

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of
Recep Avci presented on May 1, 2013

____
Sinan Kockara, Committee Chair

____
Chenyi Hu

____
Mark Smith

PERMISSION

Title: An Efficient Parallel Implementation of Structural Network Clustering in Massively Parallel GPU

Department: Computer Science

Degree: Master of Science

Recep Avci

Date

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Sinan Kockara, for the guidance and leadership he provided throughout work for master's degree. I also want to thank my dear friend, Thomas Ryan Stovall, for his great contributions to this project.

I thank the committee members, Dr. Chenyi Hu and Dr. Mark Smith, for their support.

Recep Avci

The University of Central Arkansas
May 2013

ABSTRACT

In many fields, complex networks are commonly used to represent relationships among sets of entities in real systems. Community in a network can be considered as a summary of the whole network; thus, it is a very important field. For instance, communities in a citation network might represent related papers on a single topic whereas communities on the web might represent pages of related topics. New community detection or clustering algorithms have brought us significant advances to discover hidden knowledge, to summarize the network, and to find relationships. Detecting communities in real systems has a great importance in different fields such as sociology, biology and computer science. The Structural Clustering Algorithm for Networks (SCAN) is a fast and efficient clustering technique for finding hidden communities and isolating hub and outlier nodes within a network. However, for very large networks, it still takes considerable amount of time. With the introduction of the Compute Unified Device Architecture (CUDA) by Nvidia, the scientific community has seen an explosion in applications employing graphical processing unit acceleration. In this project, we present a CUDA-based parallel algorithm, where SCAN's computation steps are carefully redesigned. We discuss transforming SCAN into a series of highly regular and independent operations suitable for acceleration via CUDA. Now, a large network or a batch of disjoint networks can be offloaded to graphics processors for quick and equivalent structural clustering. The experimental results indicate that our *parallel structural network clustering* algorithm generates exactly equivalent results to SCAN. Moreover, it is considerably faster than SCAN. Depending on the dataset, this speedup can be up to 254-fold.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1: Introduction

1.1 The concept of Network

With the advances in computer technology, the world evolved to a new globalized system. From technology to economy, this new world system is more dynamic and interactive such that entities in the world such as people and/or organizations are no longer as isolated as they used to be [1]. In the past, entities had a natural border which determined the degree of their isolation [2]. For example, a person who lives in a small village in Africa had to be content with the people who live in the same territory. However, in the new globalized system, an entity can easily interact with any other entity in the world which leads to broader and more complex interaction systems. These interaction systems are considered as *networks* in which entities are known as the *nodes* of networks. The connections between the nodes represent the relationships of these nodes.

Although the concept of network evokes computer systems, it is also used to express the interaction of the entities in real life or in any field. In the real world, a node in a network can be a person, an organization, a country, or a group while connections can be expressed as friendship, alliance, collaboration, or business partnership [3]. For instance, the social interactions of employees in a company form an internal network which may represent the company's informal structure [4]. This structure may provide certain dynamics of employees' interactions. For instance, consultations between the employees form an *advice network* in the company. When an advice exchange between two employees occurs in the company a connection of the *advice network* is formed [5].

In various fields, the network concept is used to interpret and solve many research problems either as a core or supportive idea [3], [6]. Graph theory in mathematics and computer science is a well-known field about network studies [7], [8]. In addition, some studies about the social networks in sociology [9] and some studies about complex structures in physics are examples of studies which deal with networks [10]. Generally, social scientists focus on the relationships in social networks while mathematicians and computer scientists develop methods to obtain and analyze knowledge about the structure of the relationships within the networks.

In graph theory, all networks can be represented as graphs and are denoted as G{*V*, *E*} where the graph G exists from two sets, V and E. *V* stands for the set of vertices (nodes) and *E* stands for the set of edges between the vertices. Figure 1 illustrates a simple network (graph).

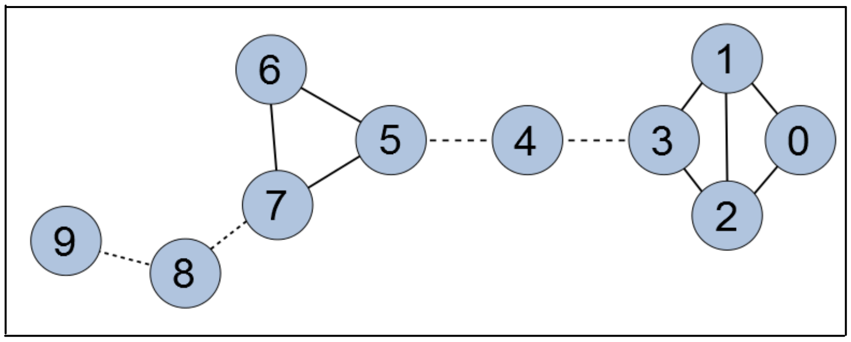


Figure 1: A simple network with10 nodes.

The edges between the vertices in a graph represent the interaction of the nodes in a network. These interactions can be one-way or two-way. We can easily understand this on Twitter which is a popular social network on the Internet. For instance, if two people follow each other on Twitter, this is an example for two-way interactions. In this case, we say there is a *directed edge* between the vertices. Directed edges are also known as *arcs*. On the other hand, if a person follows another person without being followed by that person, this represents a one way interaction but the edges are still considered as directed edges. The graphs with directed edges are called *directed graphs*. Graphs are not always supposed to have a direction between their edges. For example, an employee network in a company has undirected edges since the relationships between the employees do not have a direction. The edges in this kind of networks are called *undirected edges* and the graphs with undirected edges are called *undirected graphs*.

In graph theory, the number of connections that a vertex has is called a *degree*. The degree distribution within a network gives us information about the complexity of a network. If we divide the total number of degrees in a network by the total number of vertices, we find the *average degree* in a network.

Graph theory also tells us that the vertices in the networks can be connected to other vertices in more than one way. In other words, in a complex graph, there can be multiple edges between two vertices. Multiple connections between vertices can be understood when we think about real-world relationships. We are connected to other people in many ways. For instance, a person can be a co-worker of his/her spouse. In such a case, two people are connected with each other through two ways. Therefore, when we consider a real-world network, we cannot assume that the nodes have only a single interaction. Such an assumption causes misinterpretations on a network when we attempt to discover hidden knowledge or to detect communities within networks. Now, we will explain the community detection concept in networks in the next section.

1.2 Community Detection in Networks

The network concept, more specifically the concept of social network, has gained importance after the Internet was used in daily life. After social networks on the Internet became popular, they have partially revealed the relationships between people. Thus, we have had some ideas about the interactions of human beings up to one degree. However, they have also built complicated and discrete networks. These networks include hidden knowledge from personal information to community facts. Network clustering is one of the most common ways to detect communities within a network.

Network clustering is partitioning a network into sub-networks in which nodes have denser connections or common structures. Network clustering has other names like graph partitioning [11], community structure analysis [12], and community detection. Basically, clustering in networks is to classify nodes into different categories. We will call these categories as clusters throughout this thesis.

Community detection via network clustering reveals the bonds between the people who form the network. A well-known community detection study was conducted on a karate club by W. W. Zachary [13] in 1977. In this study, Zachary attempted to predict the split of the members of a karate club. In this karate club, there occurred a conflict between the karate instructor and the club president. The instructor demanded a raise for the lessons but the club president thought the price should not be changed and also the club administrators should give such a decision. Somehow this conflict brought up some ideological debates and, the debates polarized the club members. The problem was finalized when the club president fired the instructor. Then the instructor opened his own club and some of the members decided to join the new club. Zachary predicted members' decisions based on their relationships with only one error. Figure 2 shows how two groups were shaped after the split [14].

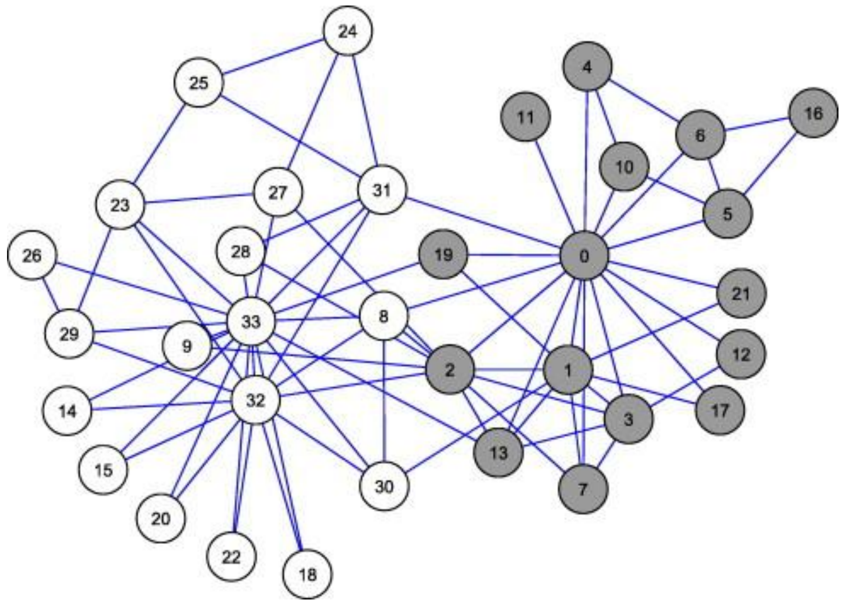


Figure 2: Zachary karate network. The nodes with darker color belong to one group; the ones with white color belong to the other group.

Zachary, in his study, revealed that the interactions between the nodes form invisible small communities. Only community detection methods can discover these hidden communities. Since the sizes of networks are very large in the new world systems, discovering knowledge is no longer as easy as it used to be and therefore, very deep and detailed studies or newly developed methods are required to obtain this hidden knowledge.

In this project, we have worked on developing a knowledge discovery method in large-scale networks, which is explained in Chapter 4. First, we will introduce the most common network topologies in the remainder of this chapter.

1.3 Network Topologies

Networks are very complicated and intricate systems. For real-world networks, it is almost impossible to classify them in terms of their structures. However, in graph theory, mathematicians and computer scientists define several network topologies according the layout of the interactions or relationships. These topologies are *point-to-point*, *ring*, *tree*, *bus*, *star* and *fully connected* [15]. Although these topologies are used to specify the shape or structure of computer networks, they are also used for real-world networks and help us to understand their structures as well.

Network topologies do not always indicate physical connection between the nodes in networks. They also indicate the interaction of the members in a social network. Understanding these topologies is essential to discover hidden knowledge within a network. For instance, most of the clustering algorithms created for networks use some parameters. To know what type of structure a network has may give a clue to determine the optimum parameters.

Social networks are usually more complicated than a specific example. Therefore, the network samples provided below should be considered as a part of a social network which is revealed based on a simple rule or relationship. We will now explain the most common network topologies listed above in the following subsections.

1.3.1 Point-to-Point

The simplest form of a network is *point-to-point* connections which is also known as *linear networks*. This kind of network has two end-points and the nodes between these end-points are connected to two other nodes. Figure 3 shows a sample of point-to-point networks.

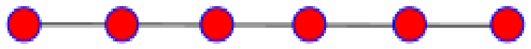


Figure 3: A simple point-to-point network

A simple example of a linear network can be seen in a family tree. Consider a woman in a family tree; she is connected to her mother and also her daughter. In the same fashion, her mother is also connected her mother and her daughter, etc. Thus, a point-to-point network is formed based on the connections of women in a family tree. In real life, these women have other interactions. This one-to-one network is not the only network in which they are connected. As stated before, we should consider this simple network as a part of more complicated networks.

1.3.2 Ring

A network with a *ring* topology is similar to linear networks except no end-points are present. In linear networks the end-point nodes are connected to only one node. On the other hand, in the ring topology, each node is connected to two other nodes such that there is no end-point. Figure 4 illustrates a simple ring network.

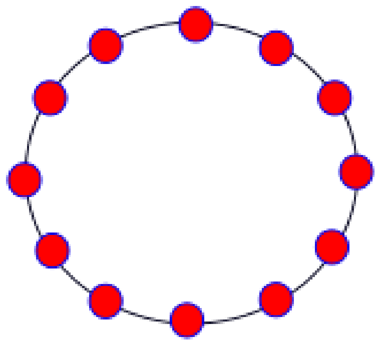


Figure 4: A simple ring network

A simple ring network can be seen in one of the popular social networks, Twitter, on the Internet. On Twitter, people can follow other people's *tweets* (statements with a maximum 140 characters) from all over the world. It is likely that, these followings can build a ring topology. In other words, a person follows another person and, that person follows another. After a number of followings the first person is followed by a person in the network, thus a ring network is formed.

1.3.3 Tree

The *tree* network is also known as hierarchical topology. In this topology, the nodes in a network are connected to higher level and/or lower level nodes. These relationships are also designated as *parent/child* relationships. A node can be connected to a parent node and multiple child nodes. A simple network sample is shown in Figure 5.

In real life, there are several samples of this kind of topology. A family tree or the hierarchical layout of a company's employees forms a tree network. From the highest level employee to the lowest level employee(s), there are interactions which form the tree network. In the same fashion, families throughout history build a tree network.

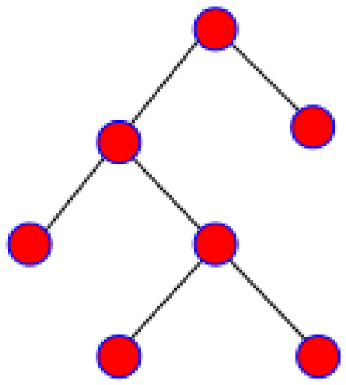


Figure 5: A simple tree network

1.3.4 Bus

In the *bus* topology, the nodes are connected to each other via a certain relationship. In other words, there is a specific bond between the nodes. In this topology, every node is connected to all other nodes via a common relationship. Figure 6 illustrates a simple example of the Bus topology.

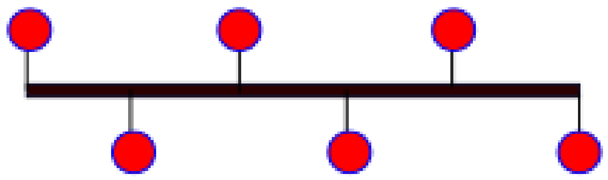


Figure 6: A simple bus network

This kind of topology is not common in real-world networks. Board members of a company or an organization can serve as a sample for this network. In such examples, all board members are connected to each other due to the positions they have in the board. In other words, the board represents the special relationships between the nodes (members).

1.3.5 Star

The *star* topology has a common point between the nodes similar to the bus topology. While in the bus topology the relationship was the common point of the nodes, in the star topology a single node is the common point. In other words, all nodes in the star topology are connected to one central node. Moreover, there are no connections between these nodes. Figure 7 shows the layout of a star topology.

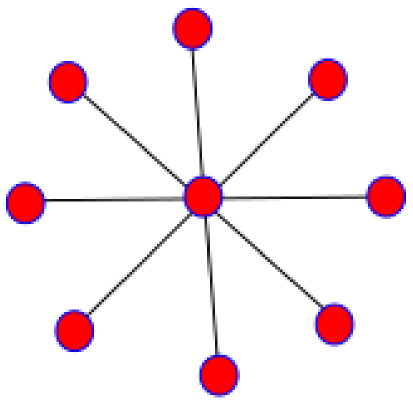


Figure 7: A simple star network

In real life, we can see networks which have star topology. Twitter again can be a good example of this kind of topology. For instance, the people who follow a certain famous person are members (nodes) of a star network. Although they do not follow each other, the person they follow is the central node in the network and forms a common relationship between the people in the network.

1.3.6 Fully Connected

In the *fully connected* topology all nodes are connected to each other. This kind of topology provides direct interaction between the nodes in the network. A node can be connected to infinitely many nodes at the same time. There is no specific limitation for the connections. Figure 8 shows a mesh topology with 6 nodes. Networks with this topology are also known as *mesh* networks. Of all topologies, this topology can grow most easily. The most complicated networks consist of mesh topologies. This is also one of the most common networks in the real world.

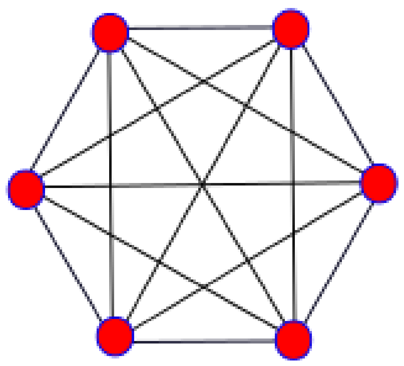


Figure 8: A simple mesh network

Any small community can be an example of this topology. In small communities, people usually have interactions with every member in the community. Therefore, their *interactions* build a mesh network.

These network topologies exist in community networks but they are usually hidden. The community detection introduced in the previous section is the one of the ways to reveal these hidden networks. The next section introduces the literature related to community detection methods.

Chapter 2: Review of Literature

Networks are high dimensional data structures. Network clustering methods target to group the nodes of these high dimensional structures into clusters. The problem of obtaining accurate clusters has been studied for years in many fields particularly in computer science, graph theory, and physics. Basic community detection in networks involves properly arranging a network structure by visual inspection. Such a method is intuitive and can only handle small networks. Hence, several network clustering algorithms have been proposed. Although these methods share some common points they are principally not the same. We will now introduce some of the common methods for network clustering (community detection).

2.1 Min-Max Cut and Normalized Cut

The min-max cut method [11] basically aims to split the network into two sub-networks. A *cut* is the total weight of edges that are removed to group the vertices into two clusters. The idea of min-max cut method is to decrease the number of connections between two clusters and increasing the number of connections within the clusters.

One of the weaknesses of min-max cut method is that there are some constraints to discover the edges removed. For instance, the size of the cluster must be similar to each other. However, this is not always true in networks, especially in social networks. There are some small groups in social networks as well as larger groups. To eliminate this problem, a new method, normalized cut [16], was introduced. The normalize cut method determines edges removed by normalizing the total number of connections between each cluster to the rest of the network.

Both methods split the network into two clusters. To obtain more than two clusters, the same methods can be applied to the obtained sub-networks. However, there is no measurement to identify if a network is split to optimum clusters.

2.2 Modularity Based Algorithms

Newman and Girvan [17] are leading pioneers who tackled the automatic community detection problem. They suggested using modularity to qualify the intensity of community structure. Their approach has been used for different applications including community structure validation and as a main function for optimization algorithms to detect communities. Thus, modularity rapidly becomes an effective method in the discovery of community structure [18]. In addition, extended work with Newman et al. and others has proven that clustering with maximizing modularity often yields promising community structure in real networks [19], [20].

Modularity is a metric introduced by Newman [17] and used for determining how good a network is partitioned. Modularity is denoted Q and calculated by the following formula.

$$Q = \sum_{c=1}^{k} \left[\frac{\mathrm{l_c}}{\mathrm{L}} - \left(\frac{\mathrm{d_c}}{2\mathrm{L}} \right)^2 \right] \tag{1}$$

where *L* is the number of edges in the graph, l_c is the number of edges between vertices within clusters *c*, and *dc* is the sum of the degrees of the vertices in clusters *c*.

When the modularity gives the value 0, either all vertices in the network are grouped into one cluster or all vertices are clustered at random. Therefore, when modularity value is close to zero we understand that a poor clustering occurred in the network. In addition to check the quality of clusters obtained, modularity can be used for optimization to find better quality clusters within a network. The studies proposed for the optimization [21] and [22] adapted the Newman's approach. Recently Newman introduced spectral optimization of modularity [23] for the optimization purpose.

2.3 Hierarchical Clustering in Networks

Hierarchical clustering is one of the most common and widely used community detection methods. It addition to group the vertices into clusters; it also gives some information about the hierarchical structure of a network. The hierarchical clustering is based on the metrics called *similarity* which is calculated for the each vertex pair in the network.

There are two kinds of hierarchical clustering: agglomerative and divisive. Agglomerative method is a bottom-up approach. The method assumes that each vertex owns its cluster. Then checking similarity values vertices are connected to each other based on their similarity values. After the each iteration, the clusters expand and the similarity values decrease for the next iteration. Thus all vertices are unified as a single cluster. Each level in the hierarchical tree can be considered as a cluster.

In the literature one can find different ways to form clusters in agglomerative clustering like single linkage and complete linkage. In the single linkage method, a vertex is added to the cluster if the similarity between it and the node at the end is greater or equal to current similarity in the iteration. In other words, similarity values between newly added vertex and other vertices are not taken into consideration. In contrast to single linkage, complete linkage requires vertices being a *maximal clique*. A *clique* is a set of vertices connected to the all vertices in the set. Only the vertices in the maximal clique can form a cluster in this approach.

The second approach in hierarchical clustering is the divisive method in which all clusters are the members of a single cluster in the beginning. Then this big cluster is divided into smaller groups at the each iteration by removing the edge which has the smallest similarity. However, it does not mean that the each iteration splits the network since there can be multiple reachable paths between vertices. Girvan and Newman's approach [24] is one of the examples of this kind. Their algorithm assumes that members in the same community should be more firmly connected rather than randomly. To split a given graph into communities hierarchically, edges consisting of the largest *betweenness* [25] which is the number of shortest paths passing through an edge are eliminated one after another.

Pons and Latapy [26] proposed the Walktrap algorithm for automatic community detection. Their algorithm adopted the idea of a random walk through a network for community detection. The main idea of this approach was that the densely connected portion of a community would tend to trap random walkers. Instead using modularity the authors introduced a similarity measure based on short walks and used it for community detection via hierarchical clustering.

Orman et al. [27] compared different community detection algorithms (Label Propagation, Eigenvector, Walktrap, etc.) with networks generated with a model from Lancichinetti et al. [28]. Normalized mutual information measure was used to access the performance of those algorithms. Walktrap has been one of the best algorithms to generate excellent results by successfully identifying communities even for high mixing coefficient values [27].

2.4 Structural Clustering

In 2007, a structural network clustering algorithm for networks strongly influenced by DBSCAN (density-based spatial clustering of applications with noise) [29], was introduced as an alternative to the Girvan-Newman based modularity algorithms [17]. Rather than using *betweenness* to partition the given network into clusters, SCAN uses the notion of structural similarity to agglomerate nodes into clusters. Structural clustering is the process of grouping members of a network into communities (clusters) based on the density of relationships (edges) among the members. The process results in a disjoint set of sub-networks which represent the hidden communities within the network.

SCAN uses the notion of structural similarity to agglomerate nodes into clusters [30]. Consider a few quantitative properties of any node in a network. The vertex structure (2) of an arbitrary node u, Γ(u), from a graph is given as the set of u and the nodes adjacent to u. Two nodes u and v have a structural similarity, σ(u,v), (3) based on the number of nodes common to the vertex structures of both nodes. These properties are summarized as follows:

$$\Gamma(u) = \{v: (u,v) \epsilon\ E\} \cup \{u\} \tag{2}$$

$$\sigma(u,v) = \frac{|\Gamma(u) \cap \Gamma(v)|}{\sqrt{|\Gamma(u)| * |\Gamma(v)|}} \tag{3}$$

$$N_{\epsilon}(u) = \{v: \sigma(u,v) \geq \epsilon\} \tag{4}$$

$$CORE_{\epsilon,\mu}(u) \leftrightarrow\ N_{\epsilon}(u) \geq 2 \tag{5}$$

The SCAN algorithm has two parameters. The first, ϵ, is the threshold structural similarity value for adjacent nodes to be considered ϵ-neighbors (4). The second, μ, is the minimum number of ϵ-neighbors a node must have to be considered a core node (5). For SCAN, nodes in a cluster must be a core node or an epsilon neighbor of a core node. When μ equals to 2, this is equivalent to the restriction that all proper clusters must contain at least one core node. We strongly believe that μ must be set to 2 because if it is set a greater value, none of the networks with point-to-point or ring topology can be identified as clusters.

Structural clustering begins with an arbitrarily chosen node v from the network. Structural similarity is calculated for each edge with *v* as an endpoint. If the node *v* has less than μ neighbors, then another node is selected. Otherwise, the node *v* is identified as a core node and a unique cluster identifier is generated. That cluster identifier is then assigned to *v* and the epsilon neighbors of *v*. This process continues until each node is visited. After structural clustering is complete, some nodes remain without membership in a cluster. Some nodes do not share enough relationships with any particular cluster and so do not merit being assigned to a cluster. If such a node bridges two or more clusters, it is then classified as a hub [31]. Otherwise, it is classified as an outlier and may be regarded as a noise. For instance in Figure 1, the node 4 is a hub while the nodes 8 and 9 are outliers.

Unfortunately, computation of SCAN for very large networks takes a large amount of time. In SCAN, many set intersection operations must be performed. Also, all nodes must be visited during graph traversal. To reduce the computation time, we introduced a GPU-based algorithm in which the necessary computations are completed by using the massively multithreaded GPU architecture. We redesigned and parallelized SCAN as a series of highly regular and independent operations in order to benefit from computational power provided by GPUs. With that, a large network or batch of disjoint networks can be offloaded to the graphics processor for quick and computationally efficient structural clustering. Before introducing our redesigned structural clustering algorithm, we will provide necessary background on general purpose computing on GPU paradigm in Chapter 3. In the first subsections of Chapter 3, we will present parallel computing concepts to better understand GPU paradigm.

Chapter 3: Parallel Computing

3.1 The Concept of Parallel Processing

Computer technology improves rapidly as a new development is released every day. However, demands from computer technology grow faster than itself, which likely triggers the developments. For instance, as computers began to be used in various areas, the amount of data obtained increased making personal computers unable to cope with them. This need has brought up the idea of parallel computing through parallel processing.

Parallel processing is executing multiple tasks concurrently on multiple processing units. A parallel application/program includes multiple active processes or threads working simultaneously to solve a problem. By increasing the number of processing units in a computer or system and providing a communication methodology between these processing units, executions can be done more efficiently in parallel.

3.2 History

The first applications of parallel computing started in 1950s. In 1960s and 1970s, with the introduction of supercomputers, parallel computing became more efficient and applicable in broader fields. The first supercomputer models use multiprocessors with a shared memory [32]. These multiprocessors work on shared data in the same architecture side by side. In the 1980s Caltech Concurrent Computation project introduced a new supercomputer built of 64 Intel 8086/8087 processors, which proved that significantly efficient performance is possible via parallel computing [33]. This system is known as the first massively parallel processors (MPPs) and was followed by several architectures. MPPs reached their peak point with the ASCI Red supercomputer by enabling over one trillion floating point operations per second in 1997 [33]. After this breaking point, MPPs have grown in size and power.

In the late 1980s, clusters began to be used in parallel computing field. A cluster is a parallel computing architecture which consists of large number of computers connected by a network. The clusters competed with MPPs for a while and, they eventually replaced most of MPPs. Today, clusters are the most common architecture used in scientific computing. Since a cluster is a collection of computers, any person can build a cluster and use it for parallel computing with a significantly lower cost comparing to supercomputers.

MPPs and clusters provided relatively stable architectures in parallel computing up to the mid 2000s. Depending on the needs and the type of applications, parallel computing architectures have evolved to purpose-specific architectures. These architectures provide a solution to the specific problem which needs computation efficiency. For example, Anton [34] is a purpose-specific supercomputer which is used for the simulations of molecular-biological systems. On the other hand, the cost of these systems does not allow them be used more widely. Therefore, this problem forced manufacturers and users to build new architectures with lower costs.

The latest trend in parallel computing is the use of multi-core processors on a single computer. This trend can be classified into two categories. The first trend is to use the multi-cores on the Central Processing Unit (CPU). Most of the latest personal computers or laptops have multi-core processors like dual-core or quad-core. These multi-core processors enable users to use these multi-cores for parallel computing. The second trend is to use graphic processing units (GPU) for computation. GPUs also have multiple cores as CPUs do. However, the ones in GPU have much more computation power. For instance, a recent Nvidia Fermi GPU can have as many as 512 cores. These cores stay idle if the GPU does not execute any graphical process. New architectures developed by the GPU manufacturers like Nvidia enable users to use GPUs also for scientific computation purposes. Tasks of a computation can be parallelized and concurrently run on the cores of GPU. Before introducing GPU-based parallel processing, we will introduce parallel processing models in the next section.

3.3 Parallel Processing Models

Processors are the main component of parallel computing. Multiple processors can be used in a computation concurrently by using parallelisms. Several classifications for the parallelisms can be found in the literature [35], [36], [37]. However, in general we can talk about two fundamental parallelisms in the models. The first one is task parallelism and the second one is data parallelism.

Some applications created in the task parallelism can be rewritten in the other way, and vice versa. However, this is not always applicable. In general, the best performance is obtained by using one of the parallel architectures considering both the structure of the application and the data set. For instance, the studies [38] and [39], are two samples which work more efficiently in the task parallelism. GPUs, on the other hand, may provide better results in data parallelism. Nevertheless, it is not easy to classify an application solely into one of two parallelisms. Sometimes using both parallelisms in the same application may provide the best output.

Based on the parallelism used, we can categorize computing systems in four groups: *single instruction single data* (SISD), *single instruction multiple data* (SIMD), *multiple instructions single data* (MISD) and *multiple instructions single data* (MIMD).

3.3.1 Single Instruction Single Data (SISD)

A *single instruction single data* model is a system with a single processor. This system works on a single data for a single instruction. Since the system executes the instructions sequentially, we cannot talk about any parallelism in this system. Most of the conventional computers have this system. Figure 9 illustrates the schema of its execution.

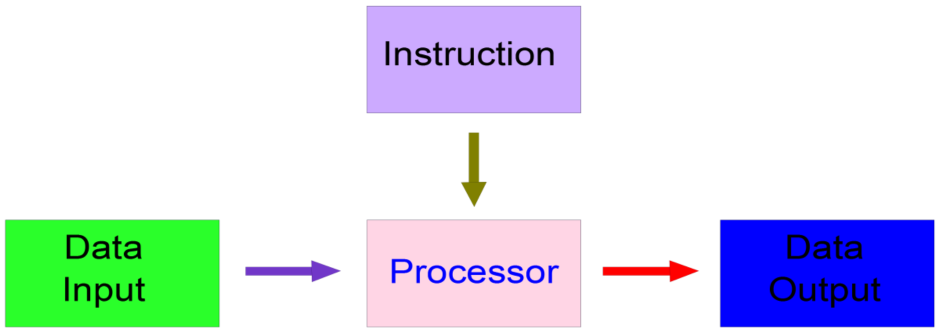


Figure 9: Single Instruction Single Data (SISD)

In this system, all instructions for the processes and all the data must be kept in the computer's memory. The performance of the execution is determined by the computer's specifications. Regular PCs, Macintosh, and Workstations are common examples for this system.

3.3.2 Single Instruction Multiple Data (SIMD)

Single instruction multiple data (SIMD) computing systems include only data parallelism. One of the examples of data parallelism can be seen in vector operations like element-wise summation or dot product between two vectors. SIMD systems execute the same instruction of the application on different parts of the data to be processed. Figure 10 shows how SIMD models work.

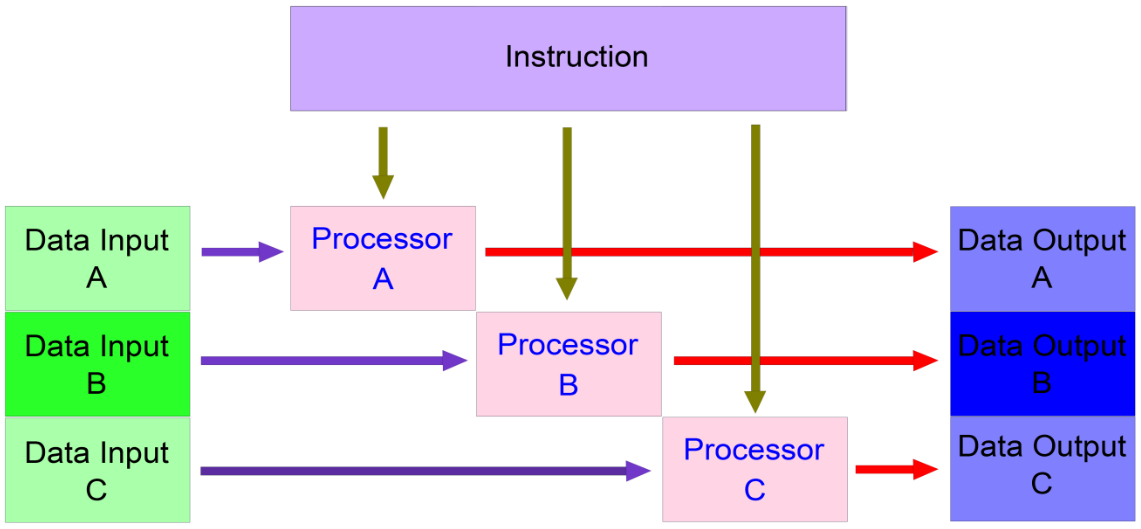


Figure 10: Single Instruction Multiple Data

This model is suitable for the scientific computing which involves a multitude of vector and matrix operations. The data is partitioned and each processing element gets a well organized portion of the data. For n processors, n different data vectors are created. Then every processor executes the same instruction on the vector they get. In this model, the way of partitioning the data plays a crucial role. This partitioning affects the performance of parallel tasks.

3.3.3 Multiple Instruction Single Data (MISD)

In the *multiple instruction single data* (MISD) model, the process of an application is divided into sub-processes which can run separately. This type of parallelism enables users to run different parts of the same application concurrently. Figure 11 illustrates the systematic of MISD models.

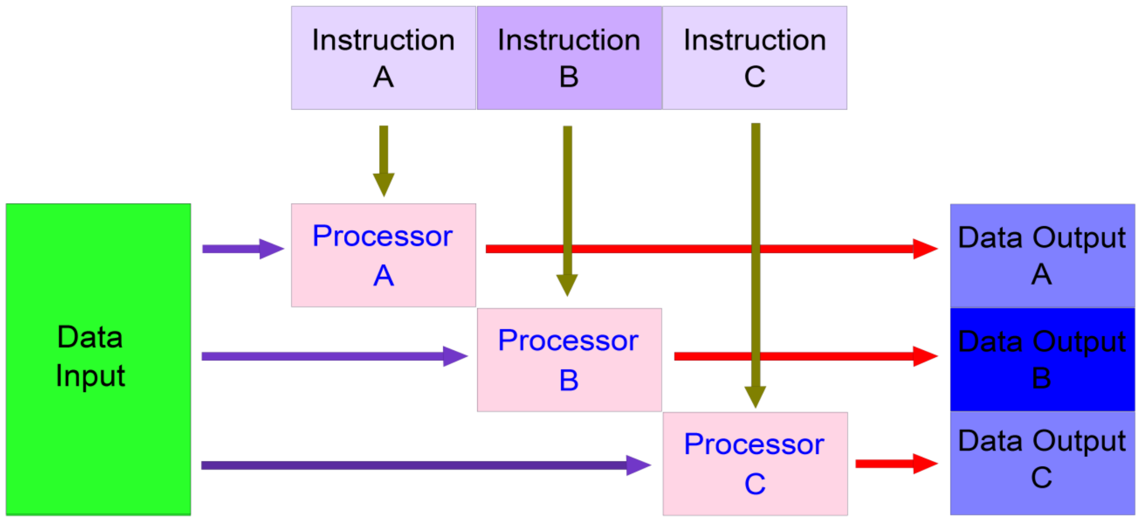


Figure 11: Multiple Instruction Single Data

The primary issue in this type is that these sub-processes must be independent from each other. In other words, one sub-process should not require any output of the other sub-processes during the execution. Each sub-process is a separate procedure in this type of parallelism. Once the application is divided into sub-processes, they are executed on the processing units for the same data. In this model, the data is used as a whole.

3.3.4 Multiple Instruction Multiple Data (MIMD)

A *multiple instruction multiple data* (MIMD) system uses both data and task parallelism in the same structure. This model can execute multiple instructions of an application on multiple data. A web server working with multi-threads is an example of this task. On such a web server, each request from same client or different clients is executed in parallel. Figure 12 shows the structure of MIMD models.

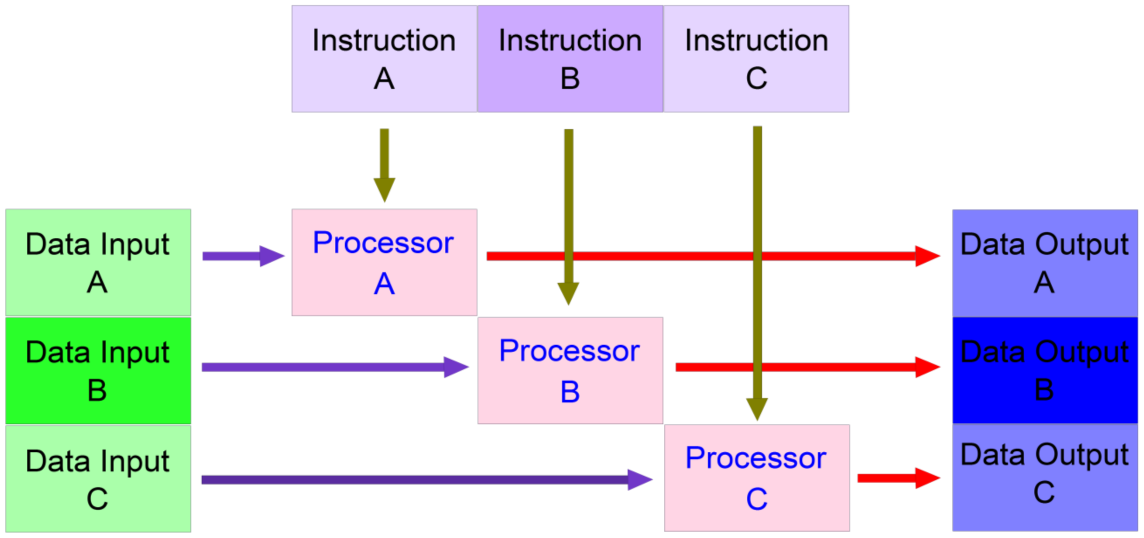


Figure 12: Multiple Instruction Multiple Data

MIMD models are categorized into two main subgroups as shared memory MIMDs and distributed memory MIMDs based on how processing units interact with the memory. In the shared memory models, there is a global memory and, all processing elements access this global memory. A communication between processing units is required for the synchronization of tasks. This communication is provided by the shared memory. After a processing unit has completed a task, the global memory is updated and the other processing units can reach modified data in the global memory.

In the distributed MIMDs each processing unit has its own memory instead of having a global memory. The processor units reach their own memory units and execute the task assigned on the data which is stored in their memory. Once a processing unit completes its job the distributed memory for that processing unit is updated. The processing units work asynchronously. If synchronization is needed between the processing units, communication is handled by an interconnection network. MIMD models are also called massively parallel processing (MPP) systems.

The shared memory systems are easy to program but they are not as extendable as distributed memory systems. Moreover, distributed memory systems can be scaled more easily than shared memory systems.

3.4 Synchronization in Parallel Processing

When multiple processes run concurrently, none can know what the other processes do and what their results are. Therefore, a communication is required whenever a process needs to reach outputs of other processes. Communication and synchronization between the tasks run by the different processes are one of the most challenging parts of parallelisms. Message passing is one of the paradigms used in the parallel systems for the communication and synchronization. Several message passing systems have been implemented around 1990s [40] e.g. Mercury/Centaur, VERTEX, The reactive Kernel, etc. Because all these message passing systems were machine-specific, it was difficult to use by a wide range of users on different machines.

Several standards were established in order to overcome portability and scalability difficulties. Thus, parallel programs can be implemented in a more practical, portable and efficient way so that they can be used widely. Message passing interface (MPI) which was introduced in the mid 1990s standardized several programming interfaces into a single standard for massively parallel processors (MPPs) and clusters [40]. MPI provides an application programming interface for the users. Moreover, communication is more efficient in MPI enabling communication without copying from memory to memory in distributed memory systems.

In late 1990s, OpenMP and pthreads emerged for the multiprocessors with a shared memory for the standardization purpose [41]. OpenMP works as a fork-join model for parallel implementations. In other words, OpenMP distributes the application as multiple tasks. After they are run on the data their outputs are joined. OpenMP controls the synchronization of the distributed tasks.

When Nvidia introduced its GPU-based parallel architecture so called compute unified device architecture (CUDA), new standards were created to make computations in parallel on the GPU. The sequential codes written in various programming languages can be executed in parallel by using these CUDA standards and run on CUDA enabled GPUs concurrently. In the next section, we will explain the GPU-based parallel computing and the CUDA architecture.

3.5 GPU-Based Parallel Computing

3.5.1 General Purpose Computing in GPU as a Co-Processor

Co-processing units have long been used to supplement the functionality of the central processing unit (CPU). Math co-processors were introduced in the 1970s to add scientific computing capabilities to word processing computers. The latest improvements in computer hardware and graphical processing unit (GPU) make possible general purpose computations on the GPU with low cost and high-speed performance. Today, GPUs provide the most computational power for the price. With Nvidia's CUDA enabled graphics cards and a proper CUDA adaptation, scientific and general purpose applications are able to harness that computational power. In the last few years, CUDA has accelerated many non-graphical applications and scientific research, with up to a few hundred times speed-up over sequential CPU execution [42], [43], [44].

Nvidia's CUDA-enabled GPUs can accelerate general purpose computing, however, mapping computation of a sequential application to the GPU architecture is non-trivial. Enabling GPUs for general purpose computing often requires careful redesign and realization of independent tasks within the sequential algorithm. One of the most difficult challenges is often utilizing high bandwidth and managing hierarchical memory. To overcome this challenge, CUDA developer must have sufficient knowledge on hardware architecture of the CUDA enabled GPUs. Before giving the details of memory hierarchy and CUDA enabled GPU hardware, we will first introduce the evolution of GPU in next section.

3.5.2 The Evolution of GPUs

In GPU's graphics pipeline, there are phases for different tasks. The pipeline receives data which represents one, two, or three dimensions as input and after processing in its phases it results as a two dimensional image. The GPU's pipeline has two type processors called vertex and fragment processors. The structure of a complete system consists of two segments e.g. central processing unit (CPU) and the GPU. Any graphical application program or any other general purpose program is incorporated into the CPU. *Figure 13* illustrates the primitive visualization throughout this architecture's processing pipeline as an example.

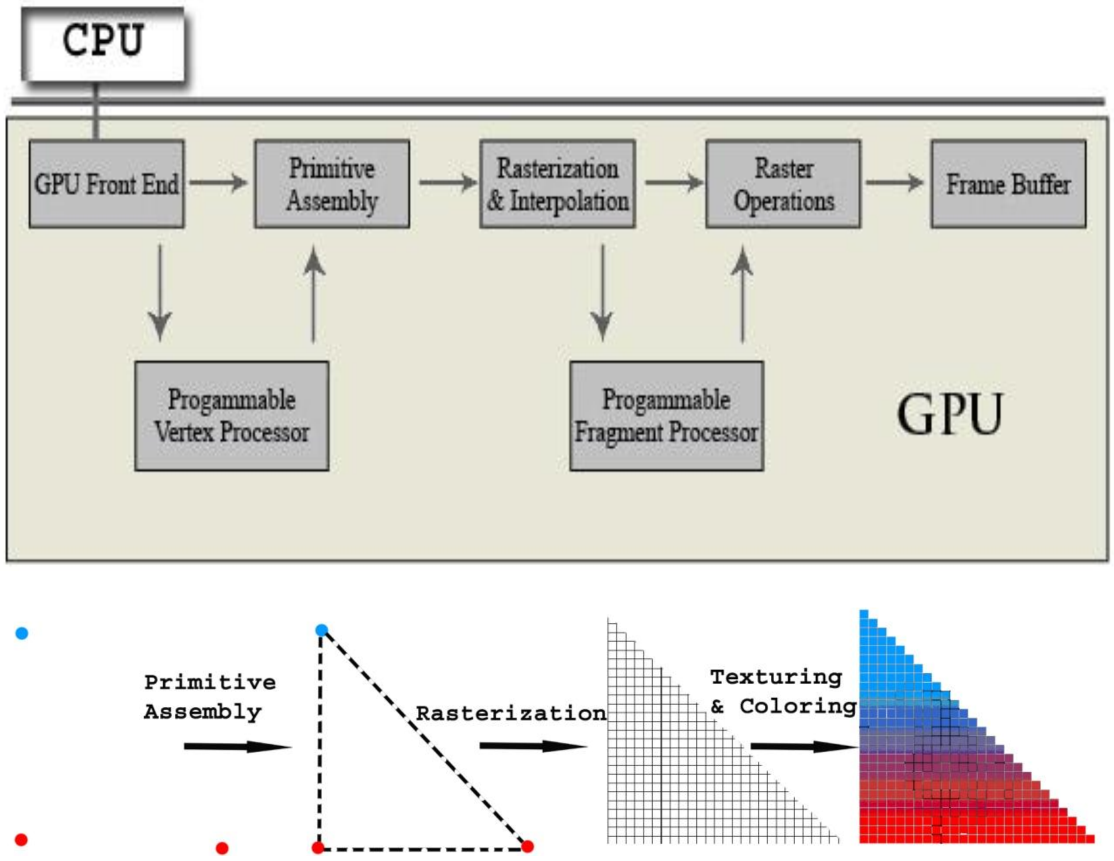


Figure 13: GPU architecture and Graphics pipeline stages

In this pipeline model, each phase receives its input from the previous phase. After processing the input, they send it to the next phase. There is a graphics memory for the access of individual stages to store intermediate computed data. For instance, in Figure 13 we see how a triangle with different colors is created via the phases in the pipeline. In the first step, three vertices of the triangle are created by vertex processors. Then rasterization and texturing are handled by fragment processors. As it is seen in Figure 13, vertex processors work significantly less than fragment processors in such a task. Since they are purpose-specific processors, they wait idle till the fragment processors are done with their jobs, which causes inefficiencies in tasks.

In mid 2000s, Nvidia introduced its unified architecture, CUDA, with more general purpose processors. These processors are unified processors that could perform vertex, geometry, pixel, and general computing operations. In other words, vertex and fragment processors were replaced by more efficient processors. Thus none of the processors stay idle while other processors are loaded with heavy work. The unified architecture brought high computation efficiency to graphical applications.

In addition to the graphical tasks, these processors can be used for parallel computing via *single instruction multiple thread* (SIMT) programming model. Now, CUDA-enabled GPUs are streaming processor units that allow parallel processing at an unprecedented efficiency. *Figure 14* illustrates the performance of GPUs comparing to CPU.

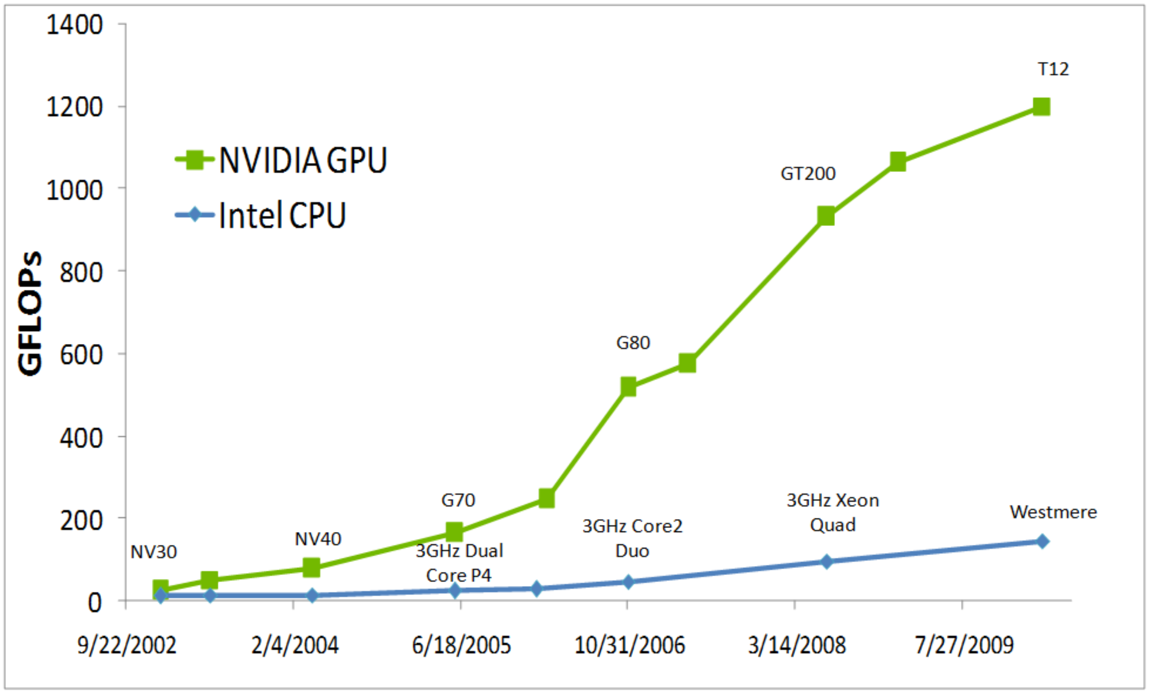


Figure 14: GPU performance chart over CPU

3.5.3 Architecture in CUDA

CUDA is a parallel computing environment introduced by Nvidia which utilizes the processing units on GPU. Nvidia introduced specific protocols to be able to use cores on GPU for computing. These protocols define the ways for CPU to use GPU in computations. In CUDA architecture, CPU is called *host* while the GPU is named *device*.

A CUDA graphics card includes a streaming processing array (SPA) which is composed of a set of streaming multiprocessors (SM) with each multiprocessor composed of a set of streaming processors (SP) called CUDA cores. Figure 15 illustrates the architecture of Nvidia G80 graphics card series. In the figure, the part in red-dashed frame is SPA in which 16 SMs exist. The part with pink color in Figure 15 shows one of these 16 SMs. In each SM, there are 8 SPs which is illustrated with light blue color. So, there are 128 processors in Nvidia G80 graphics cards which can run concurrently.

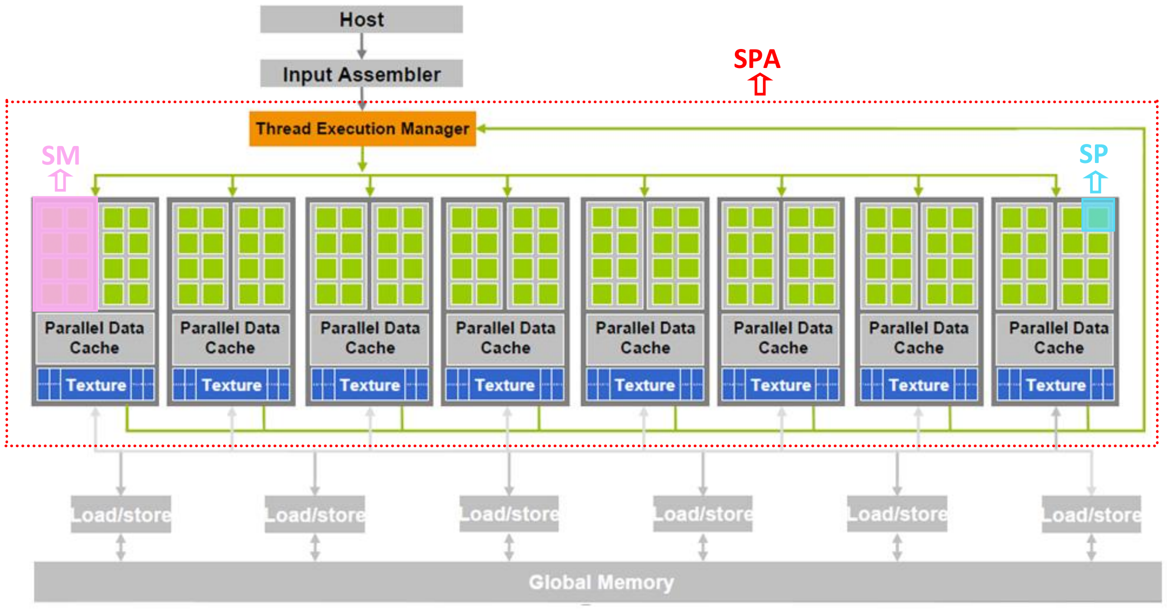


Figure 15: The architecture of streaming multiprocessors and streaming processors.

The architecture in GPU provides three main computation components. The first component is called a *grid* where is handled by a single GPU. Grids can be one or two dimensional. These grids consist of the other component, *blocks*. Every block in a grid has the same size and dimensions. Blocks include the *threads* and can have one, two or three dimensions based on the threads' architecture. Figure 16 illustrates a simple layout of the architecture.

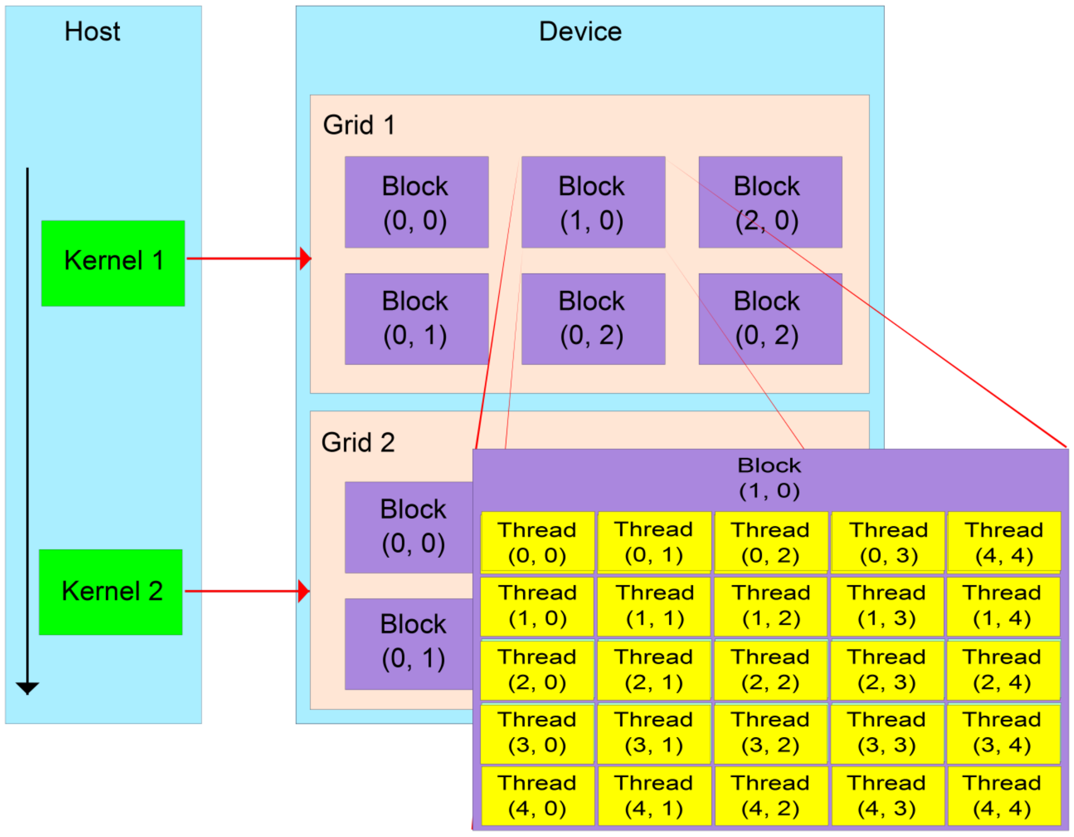


Figure 16: CUDA Architecture

Each block in the grid can have up to 512 threads. The blocks are handled by a single multiprocessor on the GPU. Today, GPUs can have up to 120 multiprocessors. Similarly, threads in blocks are handled by a single processor in the multiprocessors which handle the blocks. A multiprocessor has 8 single processors (core). Therefore a GPU can have totally the number of multiprocessors times 8 cores.

The blocks which are processed concurrently are called active blocks [45]. The threads on these active blocks execute the instructions. The active blocks consist of a set of *warps*, where a warp is a set of 32 threads executing in *single instruction multiple data* (SIMD) parallel. The warps always have the same number of threads during the execution. Figure 17 illustrates warps which work within a streaming multiprocessor.

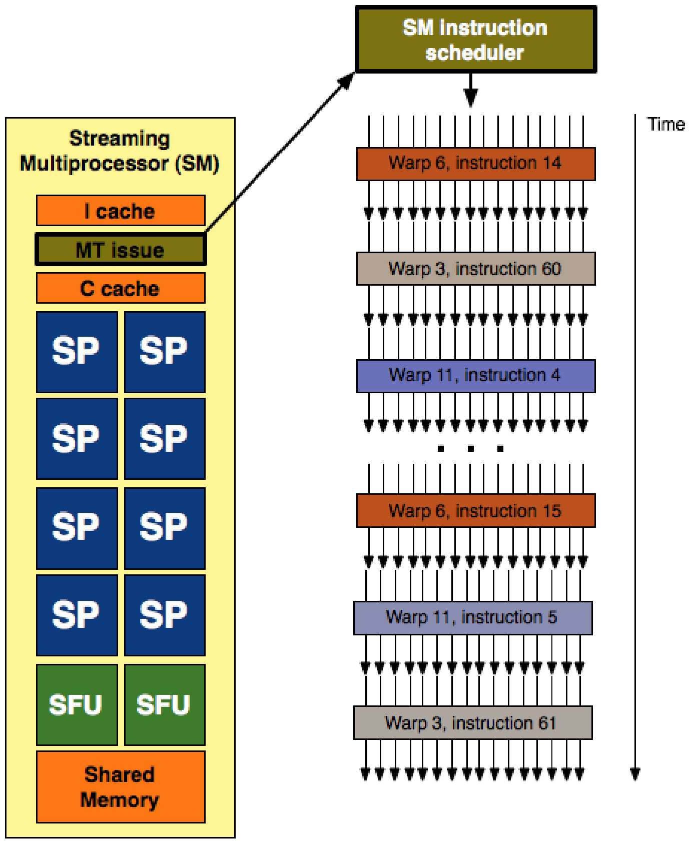


Figure 17: Warp scheduling. Threads are grouped as warps and all threads in a warp execute the same instruction on different data set. Once all threads in the warp complete their jobs, the warp gets a new instruction.

Warps are the most important part of CUDA performance. Once a task is assigned to warps, the data to be processed are stored local memory of warps and then no time is spent for the data transfer. Parallel threads work very fast and process the data very fast. The job assignments to the threads are done rapidly via warps, and thus threads do not waste time for waiting a new job assignment. Each thread in a warp executes the same instruction for different data sets. Once all threads complete their jobs, the warp gets a new instruction and thus threads keep working without wasting any time. Nvidia still hides most of the details about CUDA's execution methods, and therefore, we have limited knowledge about what is happening background.

During an execution, CUDA allows at most 8 active blocks or 24 active warps at the same time per multiprocessor (SM). Since each warp has 32 threads, this means that a maximum 768 threads can work concurrently on a multiprocessor. Depending on the number of multiprocessors the total active thread number can reach up to 23,040 in a graphics card like the GTX 285. Nevertheless, the physical limitations do not allow having that many efficient concurrent executions. Now, we will explain CUDA memory hierarchy which is very essential to improve the computation performance.

3.5.3 CUDA Memory Hierarchy

Each multiprocessor on CUDA cards has a programmable cache (shared memory) and a set of registers. Also, multiprocessors may access off-chip global GPU memory as a dynamic/static access. However, the high latency of global memory penalizes random access. Therefore, coalesced memory access (retrieving an entire block from memory) is strongly encouraged. In addition to global memory, there are also constant memory and texture memory in the architecture. Figure 18 illustrates a schema of a grid in CUDA.

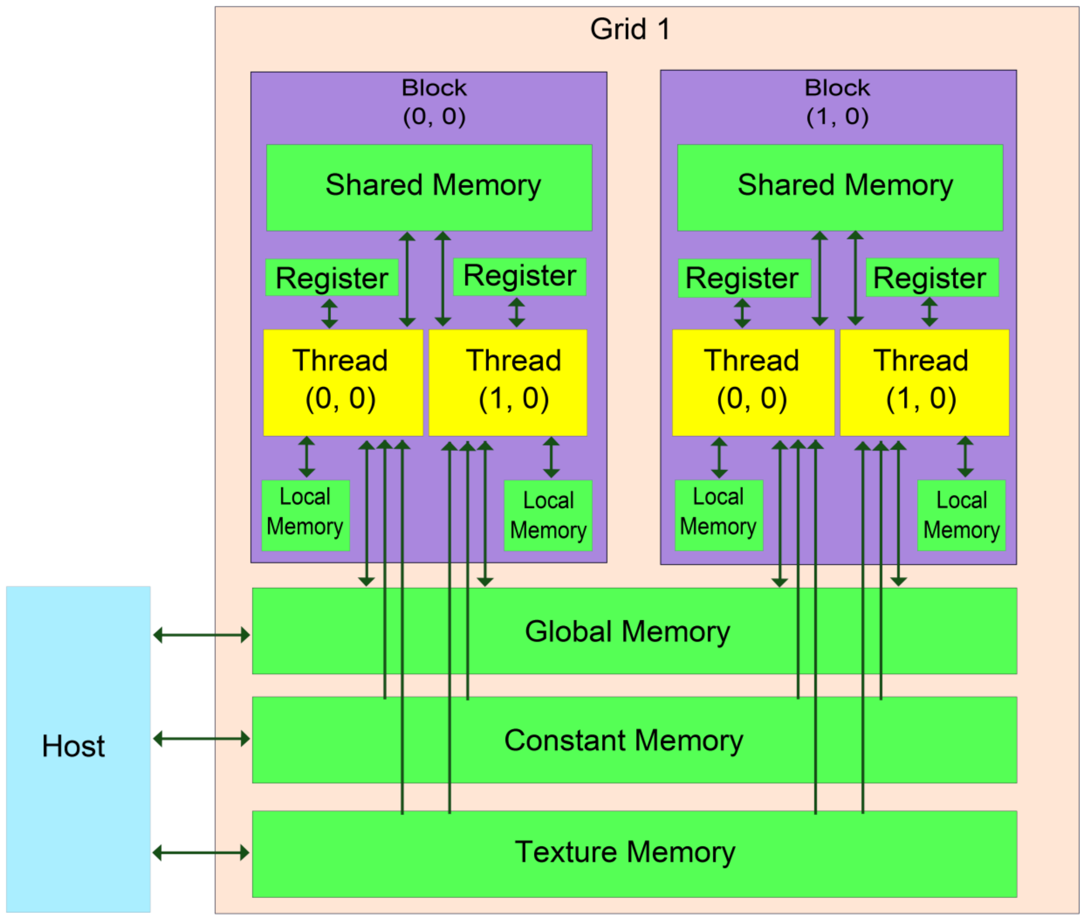


Figure 18: CUDA Memory Hierarchy

Global and constant memories enable the host to write and read by calling functions. The data which will be processed is generally passed to global memory by the host. Global memory is a part of device memory, and access to global memory by threads is handled by memory transactions. These memory transactions are limited to 32, 64 or 128 byte. In other words, only 32, 64 or 128 byte segments of the memory can be used by memory transactions. When an execution is handled by a warp, global memory allows threads in the warp do these transactions. The biggest problem in global memory is the traffic congestion when all threads attempt to access it.

The constant memory allows only read operations. Constant memory has higher bandwidth then global memory [45]. Therefore accessing to constant memory is faster than accessing global memory and can be done in a highly parallel way. Constant variables and kernel arguments are stored in constant memory.

The last memory exists in the device is texture memory. It is a memory with a hardware-managed cache. It has two dimensional spatial access patterns and only allows read operations.

In addition to device memories, there are some memories in each block too. Shared memories exist in blocks so that the threads within blocks can communicate and be synchronized. Accessing shared memory by threads is rapid. Shared memories are very helpful for threads to work coherently by sharing their results from an execution. Each thread in a block has registers allocated for itself. In contrast to shared memories, each thread can only access its registers. Registers generally store the data which is private to threads and used frequently. The data stored in shared memories and registers can be accessed in a highly efficient and parallel means.

3.5.3 CUDA Programming Model

Concurrent thread execution is handled by CUDA via kernels. A kernel is a subroutine executed by each thread in a thread batch. Threads executing the kernel are arranged primarily into a grid of thread blocks. Threads within the same block are executed on the same multiprocessor. Hence, the threads may share data and be synchronized. Threads of the same warp are *de facto* synchronized. That is, conditional branches are taken by all threads of the warp, and threads not needing to take that branch become idle until the branch completes. Therefore, the parallel algorithm must be carefully designed to minimize inefficient conditional branching.

In CUDA programming the data and kernels are passed by the host. These kernels are executed on threads using the data stored in the global memory. Depending on the kernel the output is either saved in shared memory or passed back to global memory. The execution on GPU cores continues till all kernels are executed by parallel threads.

Nvidia develops standard template libraries for the libraries implemented in different programming languages. For instance, the Thrust template library [46] is a parallel implementation of the C++ Standard Template Library for CUDA enabled GPUs. The library uses parallel primitives like parallel prefix sum and split to provide the template framework. Our implementation of *parallel structural network clustering* algorithm also uses the Thrust library extensively since procedures such as sorting can greatly increase coalesced memory access. Sorting in our algorithm is performed using radix sort from the Thrust library. Thrust partition is used for stream compaction which groups important elements of an array.

In this section we introduced GPU evolution, CUDA architecture, and programming model which was base for our parallel structural clustering algorithm. In the following chapter we will introduce our algorithm.

Chapter 4: Parallel Structural Network Clustering

The goal of constructing any GPU-based algorithm is to offload data parallel and computationally intensive pieces of the algorithm to the GPU. For our GPU-based *parallel structural network clustering*, the network is copied to the GPU device memory as an array of integer pairs, E, representing edges of the network. Figure 19 illustrates E for the sample network given in Figure 1. Once that dataset and the algorithm parameters have been sent to the GPU a series of computations are performed on the device leaving the CPU only necessary for global thread synchronization. The result is an array of integer pairs describing each nodes cluster membership or non-member classification which is fetched from GPU device memory. Algorithm 1 illustrates our implementation and its functions. Our algorithm consists of four main components. These are computation of structurally similar ϵ-neighbors, structurally connected components, clustering, and classifying non-members respectively.

Our algorithm begins once the edge list E has been copied from CPU (host) memory to GPU (device) memory. As seen from Algorithm 1, in order to create an adjacency list for each node, the undirected edge list E of length m is translated into the directed arc list A of length 2m. This is also illustrated in Figure 19. To create the array A, each parallel thread reverses a pair from E and inserts the pair into E'. Next, E and E' are joined together and copied to array A. For easy indexing of the adjacency list for each node, A is sorted using Thrust's parallel GPU sort. For instance, in Figure 19 we can see that the first two pairs of the Array A are displayed in a darker color. The second component of these two pairs forms the adjacency list of node 0 in the network from Figure 1. For computational efficiency, A is stored both as an array of integer pairs for sorting using the Thrust library, then as a pair of integer arrays for calculating set intersections. This concludes the first for loop and sort from Algorithm 1.

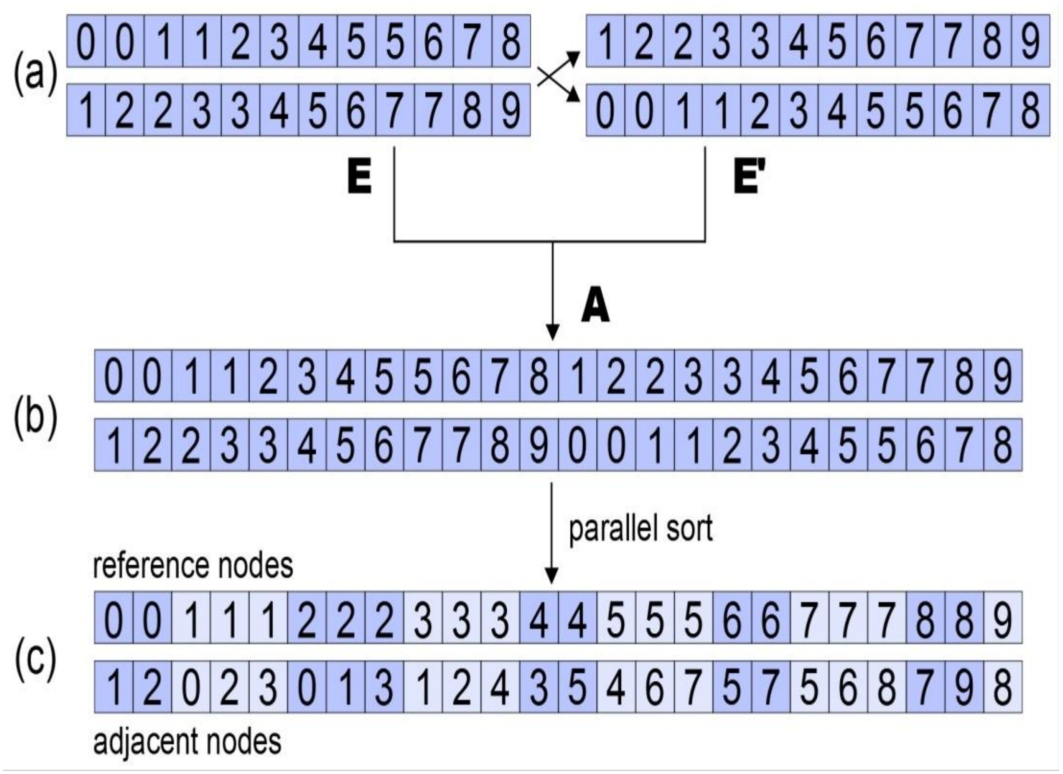


Figure 19: Forming directed arc list A, which provides the adjacency list for each node in graph G. Darker and lighter colors are used to clearly display adjacencies of each node.

In the next section, we discuss the second *for* loop of Algorithm 1 which identifies the ϵ-neighbor pairs from the edge list E. Then we will discuss computing structurally connected components followed by an explanation of clustering. Finally, the process of classifying nodes which belong to no cluster as hubs or outliers will be presented.

```
Algorithm: PaStNeC($G(V, E)$, $\epsilon$, $\mu$ )
input    :  undirected edge list $E[]$
output   :  pair list $(cid, vertexid)$
let $A[] \leftarrow \{\}, N[] \leftarrow \{\}, cid[] \leftarrow \{\}$
$isCore[] \leftarrow \{\}, parent[] \leftarrow \{\}$
for $k \leftarrow 0$ to $m - 1$ in parallel do
    $(u, v) \leftarrow E[k]$
    $A[k] \leftarrow (u, v)$ and $A[k + m] \leftarrow (v, u)$
end
sort $A$
copy $A$ to $N$
for $k \leftarrow 0$ to $m$ in parallel do
    $(u, v) \leftarrow E[k]$
    if (epsNeighbors($(u, v)$, $A$, $\epsilon$)) then
        append $(u, v)$ and $(v, u)$ to $N$
    end
end
sort $N$
for $k \leftarrow 0$ to $|V|$ in parallel do
    $isCore[k] \leftarrow false$
end
for $k \leftarrow 0$ to $length(N)$ in parallel do
    $(u, v) \leftarrow N[k]$
    count $(x, v) \in N$ where $x \neq u$
    if $count+1 \geq \mu$ then
        $isCore[u] \leftarrow true$
    end
end
$parent \leftarrow$ connectedComponents($N$)
$parent \leftarrow$ clustering($parent, isCore$)
$parent \leftarrow$ classifyNonMembers($parent, A$)
for $k \leftarrow 0$ to $|V|$ in parallel do
    let $cid[k] \leftarrow (k, parent[k])$
end
sort $cid$
output $cid$
```

Algorithm 1: Parallel Structural Network Clustering

4.1 Identifying Structurally Similar Epsilon Neighbors

Computing structural similarity requires a set intersection for each pair of nodes connected by an edge. Set intersection operations are critical for finding ϵ-neighbors. Consider the set of nodes adjacent to node 3 and the set of nodes adjacent to node 5 in Figure 1. The intersection set of the two sets of adjacent nodes provides the set of nodes adjacent to both 3 and 5; which is simply {4}.

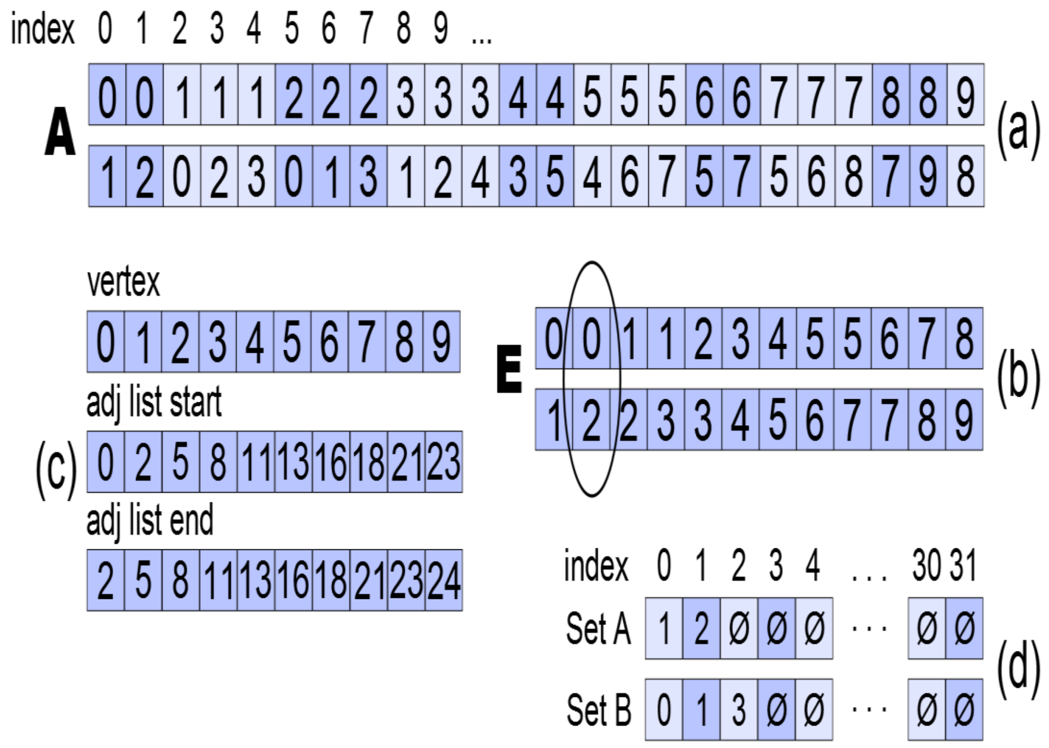


Figure 20: Consider reference nodes and adjacent nodes of A as two separate integer arrays. The boundaries for each node's adjacency list are computed using the reference node array. Here, structural similarity is computed for the edge (0, 2). Notice that Thread 1 reports a match for sets A and B, but all other threads in the warp report no match

The adjacency list for each node is extracted from the arc list A. Considering the arc list A is stored as a pair of integer arrays, the first array gives the reference node and the second array gives the adjacent nodes. Hence, the first and the last occurrence of each node in the reference nodes (see Figure 20) is recorded to give the corresponding starting and ending index positions for that node's adjacency list. These starting and ending indexes are used to calculate the structural similarity (σ) between the nodes. The following formulation shows how our clustering algorithm calculates the structural similarity for two nodes *u* and *v*:

$$adj(u) = \{v : (u, v) \in A\} \tag{6}$$

$$\sigma(u, v) = \frac{2 + |adj(u) \cap adj(v)|}{\sqrt{(1 + |adj(u)|) * (1 + |adj(v)|)}} \tag{7}$$

where adj(u) is the adjacency list of the node u. Algorithm 2 shows the steps for determining the ϵ-neighborhoods of the nodes.

Once the adjacency list is created for every node, structural similarity of each pair is computed. Each warp fetches 32 integer pairs from edge list E. To avoid branching we let all threads in the warp compute the set intersections for each pair before advancing to the next pair. Threads in the warp work together to perform a set intersection by merging sorted lists. Each thread in a CUDA kernel is given a unique thread identifier *tid*. The position in the warp, *pos* is *tid* mod 32. In Algorithm 2, the *count* variable is the number of the adjacent nodes that two different nodes share. The index of the element fetched by each thread from *SetA* is *tid/*8 and from *SetB* is *tid* mod 8. Each thread adds 1 to the variable *count* if its element from *SetA* matches the element from *SetB*. Then, if elements remain in *SetA* or *SetB*, the indices are advanced appropriately. Finally, the *count* has been computed for each of 32 pairs (due to the warp size) and in parallel each pair's count value is normalized by the geometric product of two values that are equal to the sizes of the nodes' adjacency lists.

```
Algorithm: epsNeighbors($(u, v)$, $A$, $\epsilon$)
input  : undirected edge as integer pair $(u, v)$, directed
         edge list $A$, parameter $\epsilon$
output: boolean value true if (u,v) are epsilon neighbors
let $SetA \leftarrow \{\}, SetB \leftarrow \{\}$
let $count \leftarrow 0, sim \leftarrow 0.0$
for $\forall x$ where $(u, x) \in A$ in parallel do
    append $x$ to $SetA$
end
for $\forall y$ where $(v, y) \in A$ in parallel do
    append $y$ to $SetB$
end
for $\forall x \in SetA$ and $\forall y \in SetB$ in parallel do
    if $x == y$ then
        $count = count + 1$
    end
end
$sim = (count + 2)/(|SetA| + 1) * (|SetB| + 1))$
if $sim \geq \epsilon$ then
    return $true$
end
```

Algorithm 2: Computing Structural Similarity

Based on the values calculated by each thread, *E* is compacted to the pairs which have a structural similarity value greater than ϵ. As illustrated in Figure 21, compaction happens at the pairs where their structural similarity is greater than ϵ. When this is the case neighborhood array's corresponding element becomes 1 (see Figure 21-b). Then this compacted E is concatenated with the reversed pairs of compacted E, which forms the list of ϵ-neighborhood of the nodes, N. Before continuing the connected component parts, N is sorted (see Figure 21-d).

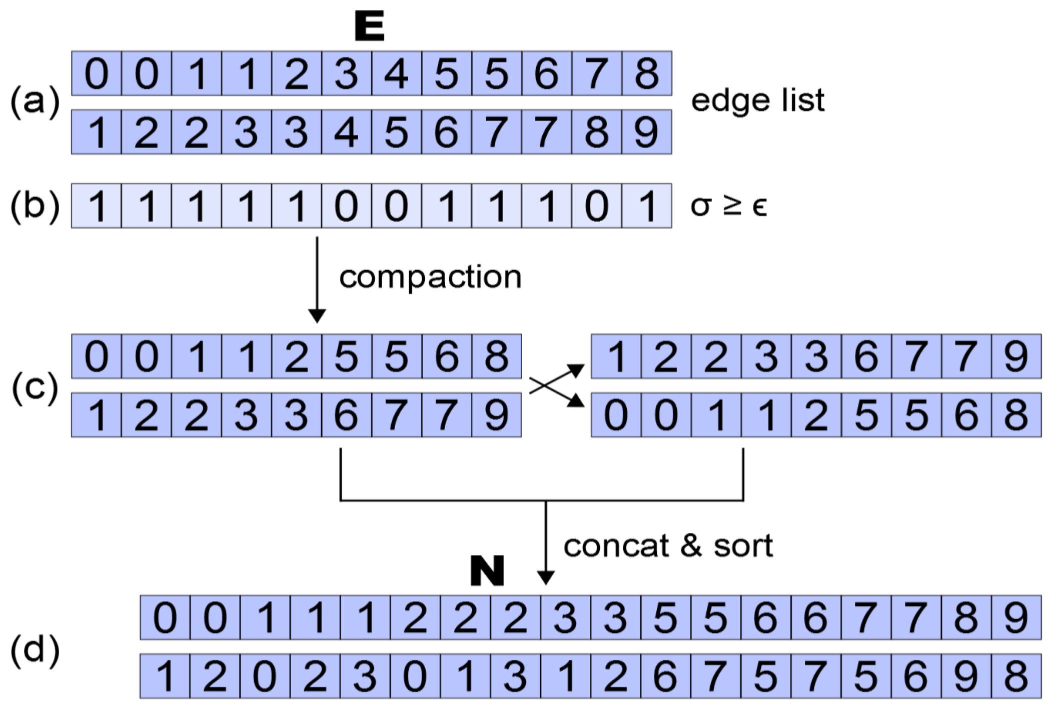


Figure 21: Edges in E with structural similarity > epsilon, are copied to a new edge list. That edge list unified with its inverse edge list forms the list of epsilon neighbor pairs N. N is sorted for computing connected components among epsilon neighbors.

4.2 Computing Structurally Connected Components

Clustering in SCAN is efficiently executed on the CPU as a breadth first graph traversal. However, breadth first search requires much conditional branching and sequential computation, so it is inefficiently executed on massively multithreaded SIMD graphics cards.

Breadth first search algorithms do exist for CUDA [47], [48], but there may actually be many small sub-graphs to be constructed in this step. Also, some CUDA implementations exist for connected component labeling [49], [50], [51], [52]; however, connected components in [49] can fail to converge. So, we provide a simple version for finding connected components adapted from [49], [53].

```
Algorithm: connectedComponents( $N$ )
input    : the set of directed edges with $\sigma \geq \epsilon$
output   : component labels represented by integer
         array $parent$
let $parent \leftarrow \{\}$
$parent \leftarrow$ linking($N$)
$parent \leftarrow$ graphContraction($parent$)
return $parent$
```

Algorithm 3: Structurally Connected Component Labeling

The connected components algorithm (see Algorithm 3) consists of two main parts. The first is called linking and the second is called graph contraction.

4.2.1 Linking

The first function of connected components algorithm is the linking phase which uses the sorted ϵ-neighbor list to assign a cluster label (parent) to nodes of a cluster (see Algorithm 4).

The ϵ-neighbor list is sorted so that each node can claim either itself or a neighbor as its parent without generating pointer cycles. Each node in the original graph is initialized as having no parent. Then, on the every iteration each node is assigned a parent from the set of itself and its ϵ-neighbors. On odd iterations the minimum node identifier is chosen as the parent. On even iterations the maximum node identifier is chosen as the parent. Alternating the direction that each node seeks a parent typically allows Linking after *r* steps rather than *d* steps, where *d* is the maximum shortest path between two nodes (or diameter) and *r* is approximately *d*/2.

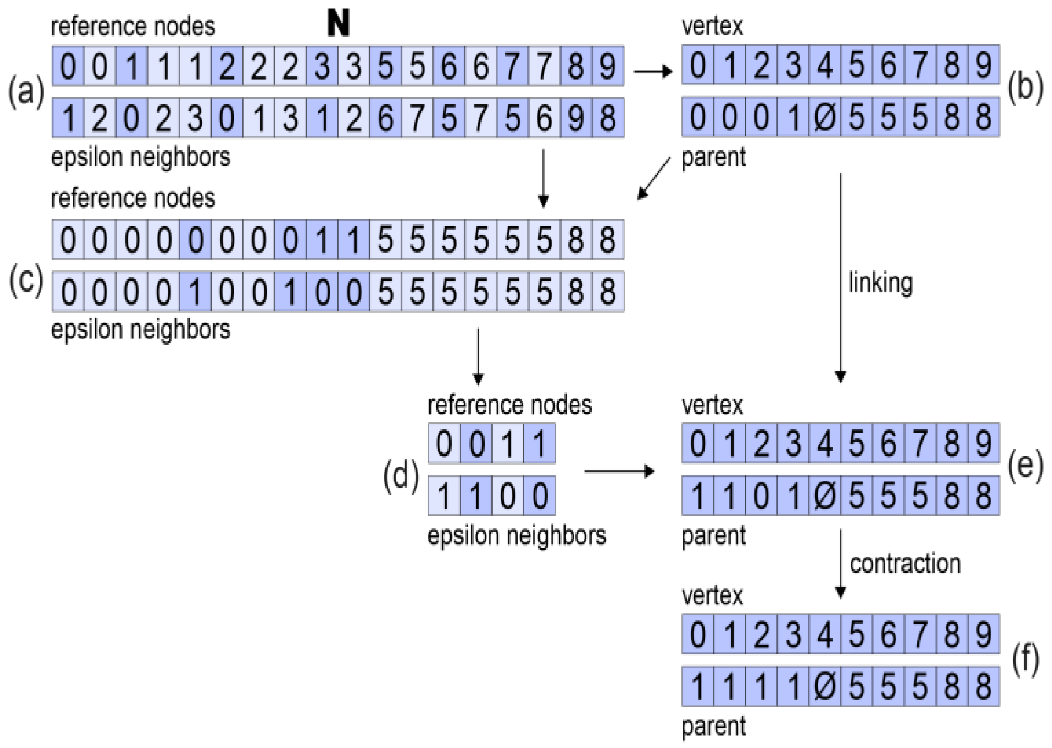


Figure 22: Computing connected components among epsilon neighbor pairs result in a set of candidate clusters.

As seen in Figure 22, the first pair of the node 1 (see N in Figure 22-a) is (1, 0), the second pair is (1, 2) and the third and the last pair is (1, 3). Considering the color differences of N in Figure 22-a, the darker colors represent the pairs which will be processed. For instance, on the first iteration the pair (1, 0) is handled and, since this is an odd iteration, the value min{1, 0} is 0. Therefore, 0 is assigned as the parent of the node 1. Similarly, the first ϵ-neighbor pair for node 2 is (2, 0) and therefore, 0 is assigned as the parent of the node 2.

```
Algorithm: linking( $N$ )
input   : the set of directed edges where $\sigma \geq \epsilon$
output  : set of tree graphs represented by integer
          array $parent$
let $parent \leftarrow \{\}$
let $i \leftarrow 0$, $s \leftarrow$ length($N$)
for $k \leftarrow 0$ to $n$ in parallel do
    $parent \leftarrow null$
end
while $s > 0$ do
    $i = i + 1$
    for $k \leftarrow 0$ to $s$ in parallel do
        $(u, v) \leftarrow N[k]$
        if $i$ is odd then
            $(x, y) \leftarrow N[k-1]$
            if $u \neq x$ or $k = 0$ then
                $P[u] \leftarrow$ min($u$, $v$)
            end
        else
            $(x, y) \leftarrow N[k+1]$
            if $u \neq x$ or $k = s - 1$ then
                $parent[u] \leftarrow$ max($u$, $v$)
            end
        end
    end
    sort $N$;  for $k \leftarrow 0$ to $s$ in parallel do
        $(u, v) \leftarrow N[k]$
        $N[k] \leftarrow (parent[u], parent[v])$
    end
    for $k \leftarrow 0$ to $s$ in parallel do
        $(u, v) \leftarrow N[k]$
        if $u = v$ then
            remove $(u, v)$ from $N$
        end
    end
end
```

Algorithm 4: Linking

At the end of the each iteration, the nodes in the list of ϵ-neighbors are replaced by their *parent*. For instance, in Figure 22-c the node 2 has changed to its parent which is 0.

In the same fashion, the node 3 is replaced by its parent, node 1. As illustrated in Figure 22-c, after all replacements are made N is updated using Thrust partition to effectively remove pairs containing identical nodes. N is then sorted (see Figure 22-d). We can see that the remaining pairs are (0, 1) and (1, 0). Since this is the second iteration and it is an even iteration, the maximum value of the pairs (in this case 1) is assigned as the parent of nodes 0 and 1. The updated parent list can be seen in Figure 22-e. Notice that after linking is done, the parent of the nodes 0, 1, and 3 is the node 1 while the parent of the node 2 is 0. Linking terminates once N is empty.

The second critical part of connected component algorithm is graph contraction which will be presented next.

4.2.2 Graph Contraction

This phase is to guarantee that all nodes in the same connected component have the same parent and is accomplished using the method of pointer doubling. Pointer doubling is a method where each node's parent is replaced by the parent of the node's parent and results in all nodes of a tree having the same parent.

As seen from Algorithm 5, each thread k fetches the parent of node k as x. If x is not null, then the thread fetches the parent of x as y. If x and y are different, then y is stored as x, the parent of node k. This continues for each thread until x equals to y. For example, in Figure 22-e, the parent of node 2 is 0 after the linking is done. In graph contraction the appropriate thread traverses the hierarchy of parents for node 2 to the top level parent. First it checks the parent of the node 2 which is 0. Then it checks the parent of 0 which is 1. Since the parent of node 1 is itself, node 1 is a top level parent, and the thread ends the traversal. The parent of 2 is now updated as 1. Thus, the thread has completed graph contraction for node 2.

```
Algorithm: graphContraction( $parent$ )
input    : set of tree graphs represented by integer
           array $parent$
output   : component labels represented by integer
           array $parent$
let $notDone \leftarrow true$
while $notDone$ do
    $notDone \leftarrow false$
    for $k \leftarrow 0$ to $n$ in parallel do
        $x \leftarrow P[k]$
        if $x \neq parent[x]$ then
            $notDone \leftarrow true$
            $parent[k] = parent[x]$
        end
    end
end
return $parent$
```

Algorithm 5: Graph Contraction

Graph contraction concludes with a disjoint set of structurally connected components, however, the components must be verified as clusters in the next step of our algorithm, called clustering.

4.3 Clustering

Not all structurally connected components are considered proper clusters. In our algorithm, μ is set at μ=2, and so a proper cluster is a structurally connected component containing at least one core node. In the clustering process components which do not contain a core node are removed from the set of structurally connected components. Before clustering, the number of ϵ-neighbors is counted for each node to determine which nodes are cores (see Algorithm 1). Nodes with at least μ ϵ-neighbors are cores. As illustrated in Algorithm 6, thread k fetches whether the node k is a core node or not. If node k is a core, then x, the parent id of node k, is stored in array clusters[x]. Next, each node's parent id x is replaced by clusters[x]. Hence, the parent ids of nodes which are not members of proper clusters become null. Finally, each node's parent id becomes its cluster id.

```
Algorithm: clustering( $(parent, isCore)$ )
input   : component labels represented by integer
          array $parent$
output  : cluster labels represented by integer array
          $parent$
let $clusterRep[] \leftarrow \{\}$
for $k \leftarrow 0$ to $|V|$ in parallel do
    let $clusterRep[k] \leftarrow null$
end
for $k \leftarrow 0$ to $|V|$ in parallel do
    let $p \leftarrow parent[k]$
    if $isCore[k]$ then
        let $clusterRep[p] = p$;
    end
end
for $k \leftarrow 0$ to $|V|$ in parallel do
    let $p \leftarrow parent[k]$
    $parent[k] \leftarrow clusterRep[p]$;
end
```

Algorithm 6: Clustering involves removing false clusters from the candidate clusters.

Figure 22 shows that after the connected component part is executed the nodes 8 and 9 are assigned to a parent id of 8 (see parent in Figure 22-f). Although nodes 8 and 9 form a structurally connected component neither node is a core. Therefore, they cannot represent a cluster. After the clustering part is completed we can see that we have only two clusters e.g. clusters 1 and 5 (see parent in Figure 23-b). Nodes with a null parent are considered non-members. The next step after clustering is classifying non-members as either hubs or outliers.

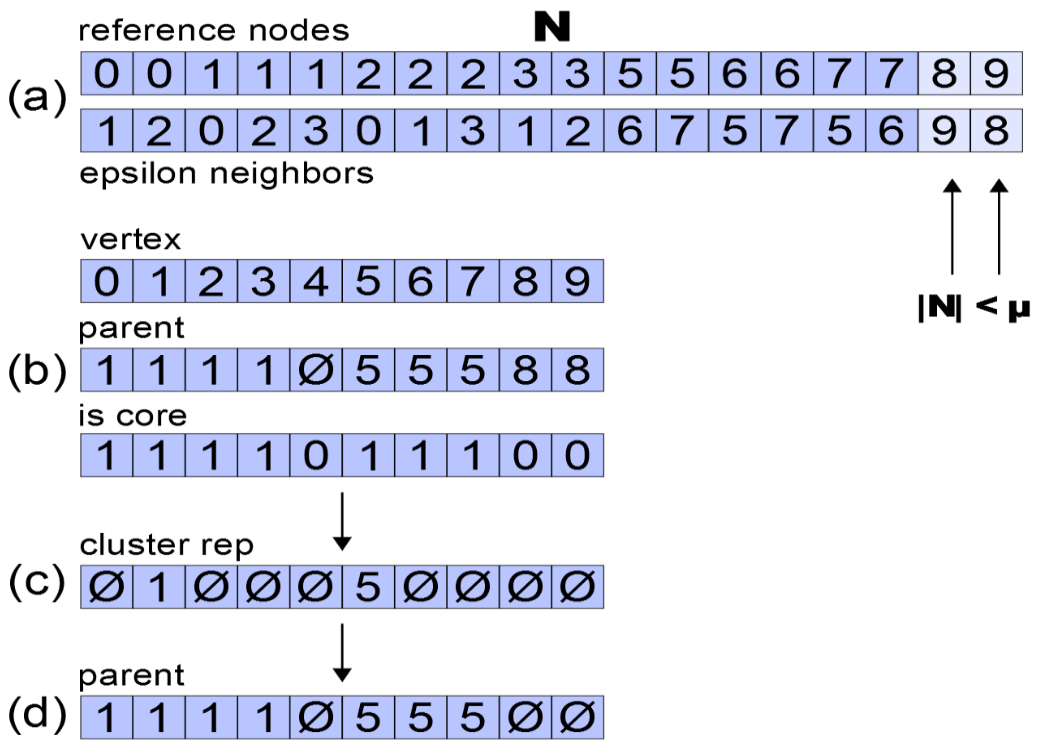


Figure 23: Nodes in graph G with μ or more entries in ϵ-neighbor pairs list N are considered core nodes. The parent nodes of cores are promoted to cluster representatives. Values of cluster representatives are then gathered into parent by parent keys.

4.4 Classifying Non-Members

The final step of our algorithm is to classify the nodes in the network which remain without a parent after the algorithm has performed clustering. How these non-member nodes are further classified as hubs or outliers is illustrated in Algorithm 7.

```
Algorithm: classifyNonMembers($parent$, $A$)
input  : cluster labels $parent$, adjacency lists $A$
output : node clusters and classifications as integer
         array $parent$

for each pair $(u, v)$ in $A$ in parallel do
    if $parent[u]$ is null then
        let $v \leftarrow parent[v]$;
    else
        let $u \leftarrow null$; let $v) \leftarrow null$;
    end
end
for each pair $(u_k, v_k)$ in $A$ in parallel do
    if $u_k = u_{k-1}$ and $v_k \neq v_{k-1}$ then
        let $parent[u_k] = HUB$;
    end
end
for each vertex $v$ in $G(V, E)$ in parallel do
    if $parent[v]$ is null then
        let $parent[v] = OUTLIER$;
    end
end
return $parent$
```

Algorithm 7: Nodes not belonging to a cluster are classified as either hub or outlier node.

Classifying non-members begins with removing the adjacency lists of cluster members from arc list A, the set of adjacency lists. This is implemented using the parallel partition function provided by the Thrust library to move the adjacency lists of interest to the beginning of A. Then, for each non-member reference node in A (see darker color nodes in Figure 24-a), the adjacent node's id is replaced by the cluster id of that adjacent node. Next, pairs in A which have null cluster id in the adjacency node position are removed from A. Finally, each cluster id in the adjacency node position s of

A is compared to the cluster id at (s-1). If the two are different and the reference nodes at s and s-1 are the same, then the cluster id of that reference node is changed to identify the node as a hub.

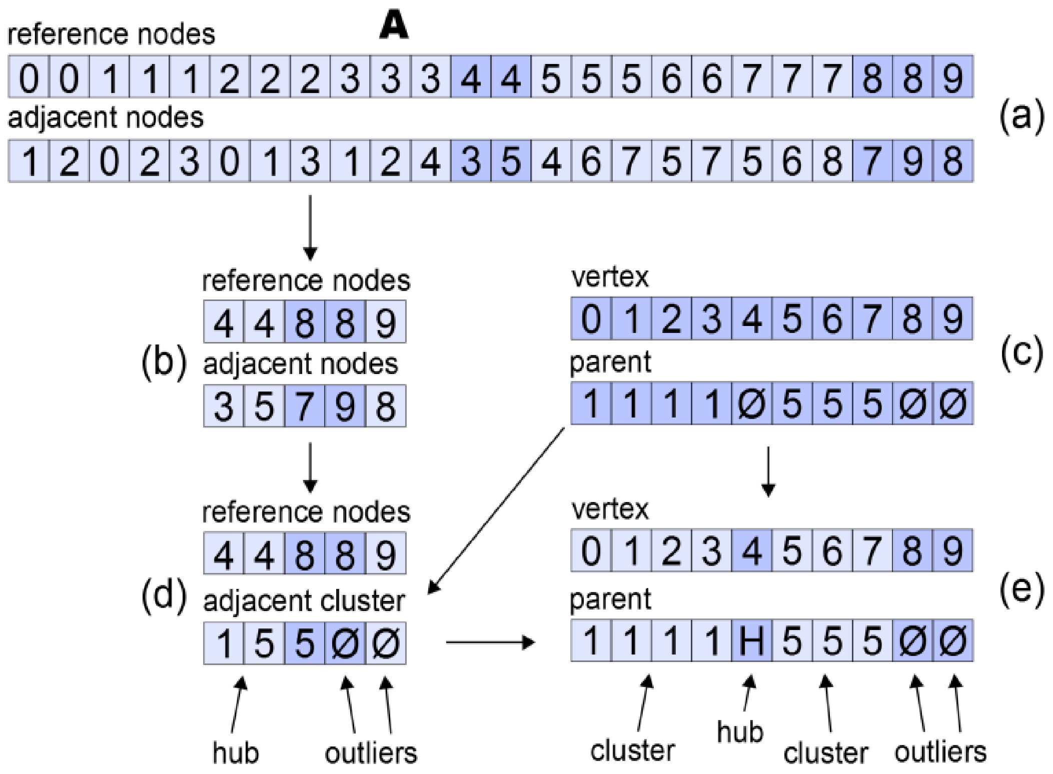


Figure 24: Pairs in arc list A with non-members as the reference node has the adjacent node replaced by the parent of the adjacent node. Non-member reference nodes with more than one adjacent cluster are identified as a hub node. Otherwise, the non-member node is an outlier.

For example, the adjacent nodes of the node 4 are 3 and 5. The cluster ids of the nodes 3 and 5 are 1 and 5 respectively. In other words, the adjacent clusters of the node 4 are 1 and 5. Since the node 4 has two different cluster ids, it is considered as a hub. Once this step is complete, all non-members which are not hub nodes are considered to be outlier nodes e.g. the nodes 8 and 9.

Our algorithm has now identified each node’s cluster membership or non-member classification, but the resulting array parent must be further processed for the output to be presentable (see Algorithm 1). If node u belongs to a cluster the cluster identifier is given as parent[u]. Otherwise, parent[u] is null for outliers and non-null for hub nodes. In parallel, each thread k inserts the integer pair (k, parent[k]) into the array clusterID at position k. Then, clusterID is sorted by the parent component and transferred to host memory. When that is accomplished the clustering algorithm is finalized.

Chapter 5: Results

Implementation of the parallel algorithm was written in C++ for CUDA and executed on the commodity graphics processor Geforce GTX 460. The results of our parallel algorithm are compared with SCAN's results. The sequential version is performed on the Intel Core i5 processor. We first tested our parallel algorithm over real world data which was also used in original SCAN paper e.g. NCAA and political books. In order to display exact equivalence of results generated from both SCAN and our algorithm we generated bitmap images of the adjacency matrix of two real datasets sorted by cluster id and vertex id.

To test the performance we have generated random networks using the GTgraph graph generator suite [54]. GTgraph is a synthetic network generator introduced by Bader et al. and is capable of generating networks with different characteristics. In our experiments we have used SSCA#2 networks and RMAT networks.

RMAT networks have a large number of vertices. They also have a small degree for the most of the vertices while a few vertices have a large degree. This model is the closest representation of the large real-world-networks like social networks. SSCA#2 graphs are made of random sized *cliques* of vertices with a hierarchical distribution of edges between *cliques* based on a distance metric. A *clique* is defined as a maximal set of vertices where each pair of vertices is connected by directed edges in one or both directions.

5.1 Equivalence to SCAN

We use two real world networks to compare the clustering of SCAN and our parallel version. For each cluster found in the network, the members are sorted by a node identifier and given the least node identifier in the cluster as their cluster identifier. Next, the set of nodes is sorted by cluster identifier. Then, the sorted list of hub nodes is appended followed by the sorted list of outlier nodes. Using this ordering, the adjacency matrix for each graph is built and printed as a black and white bitmap. Each pixel at row i, column j in the bitmap represents a link from the node in the list of nodes at position i to the node at position j. These bitmaps are given in Figure 25.

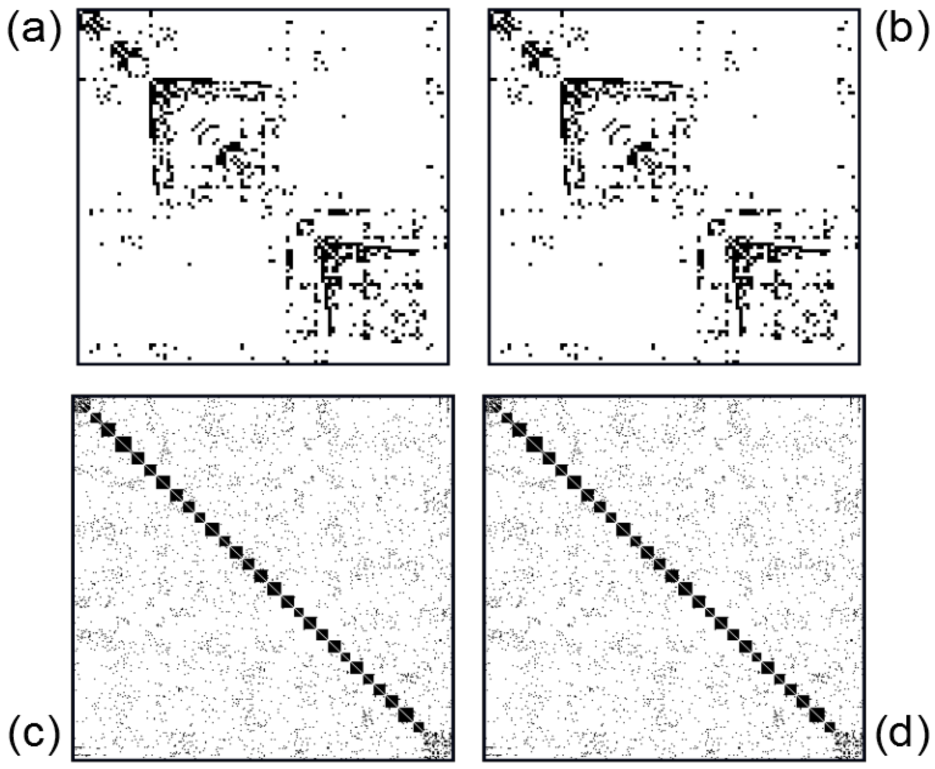


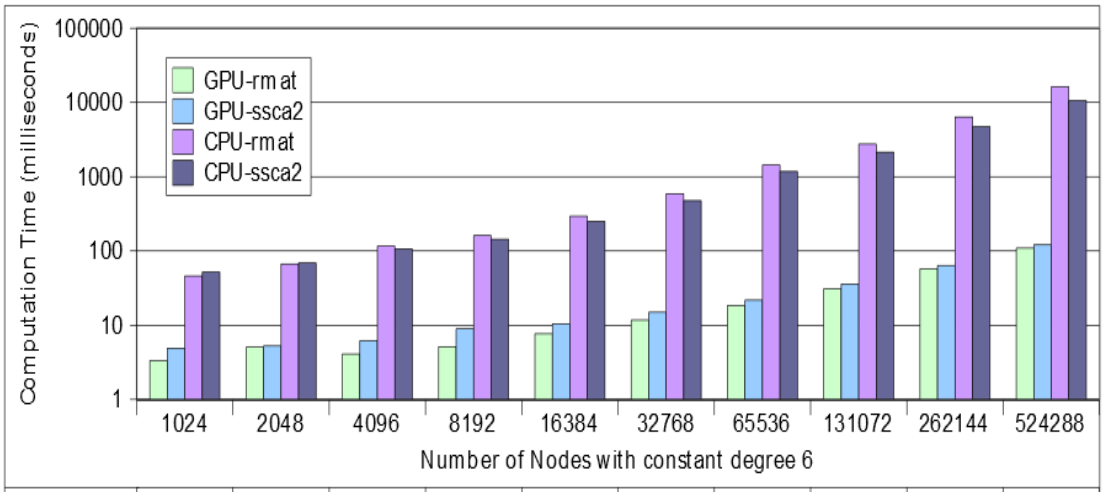
Figure 25: Bitmap representations of the adjacency matrix for two real world data sets. (a) Adjacency matrix of Political Books clustered by SCAN. (b) Political Books clustered by the parallel version. (c) NCAA teams clustered by SCAN. (d) NCAA teams clustered by the parallel version. Bitmaps for Political Books clustering has been scaled up to fit.

The first real world network is a set of 105 books about U.S. politics. The nodes represent political books sold through Amazon.com. The edges represent the books frequently bought by the same buyers. Clustering this network with SCAN using $\epsilon = 0.4$ and $\mu = 2$ produces three clusters. Naturally, these clusters are sets of liberal, conservative, and neutral biased books. As illustrated by the adjacency matrix bitmaps in Figure 25, computing this clustering with our algorithm using the same parameters produces the exact same results.

The other real world network is a football game schedule of the National Collegiate Athletic Association (NCAA). The 323 nodes in the network represent teams that play in the NCAA or against those teams. Each edge represents a game scheduled. For instance, for the parameters ($\epsilon = 0.5$, $\mu = 2$), SCAN produces 29 dense clusters, possibly representing sub-conferences. Using those same parameters, our parallel version again produces the exact same clustering as SCAN.

5.2 Performance

Clustering of any kind generally involves two basic tasks: computing similarity metric and generating clusters based on that metric. We discuss the performance of both tasks computed together. In addition, we also show the performance of our algorithm supposing the similarity has been pre-computed. As illustrated in Figure 26, we have generated 10 SSCA#2 networks and 10 RMAT networks. In both groups the number of vertices of the networks starts from 1024 and increases as the power of two.



GPU-rmat	3.3	5.0	4.1	5.1	7.7	11.7	18.5	31.0	57.5	109.1
GPU-ssca2	4.9	5.3	6.2	8.9	10.5	14.9	21.8	35.9	63.7	121.0
CPU-rmat	46	67	116	161	294	591	1436	2753	6387	16273
CPU-ssca2	52	69	106	144	250	476	1176	2145	4722	10599

Figure 26: The bar graph and table of the results for the R-MAT and SSCA2 networks with a fixed average of degree of 6.

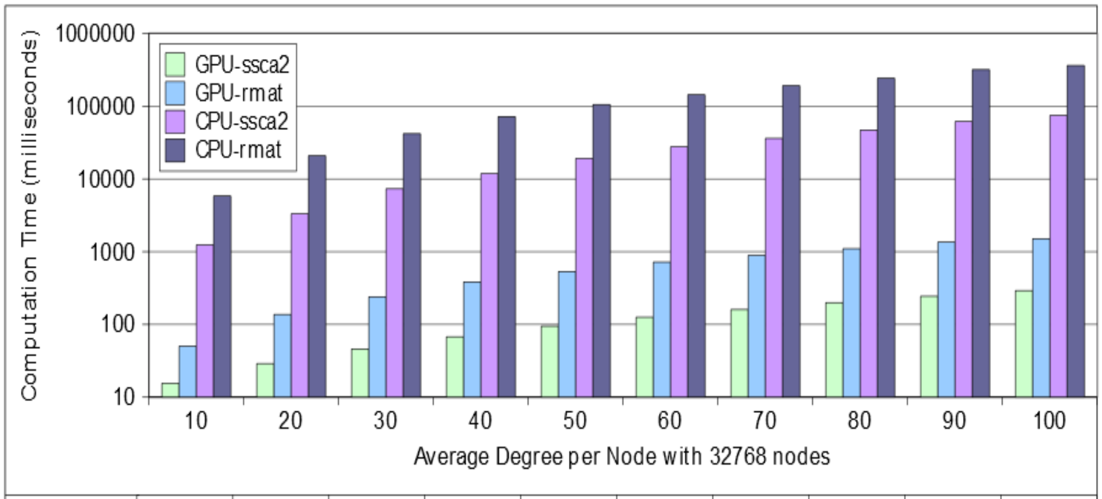
Figure 26 shows the results for the networks tested on both the SCAN and our algorithm. The computation time includes the process of calculating the structural similarity values. For each network, it can be seen that the parallel version we implemented has a much lower computation time. For instance, the computation time of the SCAN for 1024 edges is 46 milliseconds while the computation time of the parallel version is only 3.3 milliseconds for the RMAT network. In other words, our algorithm is nearly 14 times faster than the SCAN for 1024 vertices. We observe almost the same speedup for the SSCA#2 network. As the number of vertices increases the parallel version is getting significantly faster than the SCAN. For the RMAT network with 524,288 vertices, the computation time of the SCAN is 16,273 milliseconds while it is only 109.1 milliseconds for the parallel version. Our algorithm is 149 times faster than the SCAN for the RMAT network with 524,288 vertices. Table 1 shows the speedup ratio of the computation times of both the SCAN and the parallel version.

Average Degree 6			**32,768 Nodes**		
Nodes	**RMAT**	**SSCA#2**	**Degree**	**RMAT**	**SSCA#2**
1024	13.8	10.6	10	115.3	79.5
2048	13.3	13.1	20	151.3	114.2
4096	28.3	17.1	30	173.6	158.1
8192	31.3	16.1	40	185.9	173.4
16384	38.3	23.9	50	196.6	201.8
32768	50.5	31.9	60	198.6	218.5
65536	77.8	54.0	70	214.2	224.7
131072	88.7	59.7	80	219.6	235.0
262144	111.1	74.1	90	235.0	250.5
524288	149.2	87.6	100	239.2	254.2

Table 1: Speedup rates for the graphs with a constant average node degree (Left), and for the graphs with a constant number of nodes but varying average node degree (Right).

The performance results imply that the speedup of the parallel version will be much higher for the networks that have much higher number of vertices. Since the serial version of SCAN does not work for larger networks (e.g. larger than 524,288) we did compare our algorithm and SCAN up to these number of nodes. However; with the 1GB of memory available on the Geforce GTX 460, our algorithm can run e.g. on up to 16,000,000 edges. If greater memory capacity GPUs or multiple GPUs are used for the computation, the parallel algorithm is easily scalable.

In a network or graph, degree is defined as the ratio of the number of the vertices divided by the number of the directed edges. In the previous experiments, we have used networks that have an average degree of 6. In this experiment set, we have used a fixed number of vertices (32,768) with 10 different degrees ranging from 10 to 100.

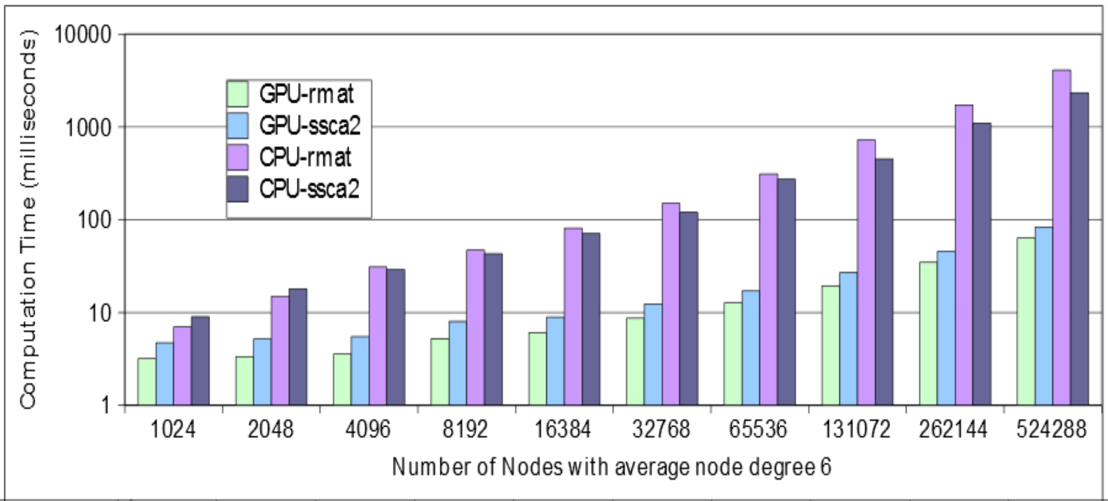


GPU-ssca2	15.6	29.0	46.2	67.8	94.6	125.7	160.3	199.4	244.2	292.0
GPU-rmat	50.3	137.2	240.1	384.8	534.2	719.0	889.0	1098.6	1347.2	1507.9
CPU-ssca2	1238	3309	7307	11759	19100	27462	36035	46860	61161	74220
CPU-rmat	5807	20756	41677	71518	105043	142800	190432	241296	316601	360634

Figure 27: The bar graph and table of the results for the R-MAT and SSCA#2 networks with a fixed numbers of vertices of 32768.

Figure 27 shows the results of both the parallel version and the SCAN. The computation time of the SCAN for the degree of 10 is 5,807 milliseconds while the computation time of our algorithm is only 50.3 milliseconds for the RMAT network. Thus, our implementation is approximately 115 times faster than the SCAN for 32,768 vertices with the degree of 10. For the RMAT network of 32,768 vertices with the degree of 100, the computation time of the SCAN is 360,634 milliseconds while it is only

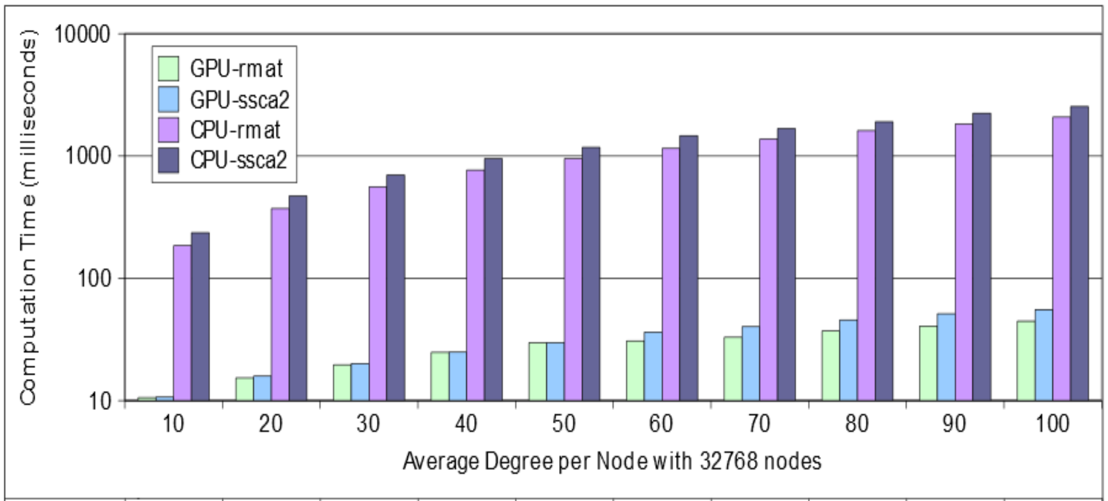
1,507.9 milliseconds for the parallel version. Hence, our algorithm is 239 times faster than the SCAN for the RMAT network for 32,768 vertices with the degree of 100 and even greater for SSCA#2 networks with the same dimensions.



GPU-rmat	3.2	3.4	3.6	5.2	6.1	8.7	12.7	19.4	34.7	63.5
GPU-ssca2	4.7	5.2	5.5	8.0	8.9	12.4	17.1	26.9	45.6	83.4
CPU-rmat	7	15	31	47	81	150	311	723	1724	4102
CPU-ssca2	9	18	29	43	71	120	273	452	1102	2332

Figure 28: The bar graph and table of the results for the R-MAT and SSCA2 networks with a fixed average of degree of 6. The computation times do not include the process of calculating the structural similarity.

When the computation of the structural similarity is not included in the computation time our algorithm is still significantly faster than the SCAN. For 1,024 vertices with the average degree of 6, the parallel version is 2.2 times faster than the SCAN for RMAT networks and 1.9 times faster for SSCA#2 networks. As the number of vertices increase, the speedup rate reaches almost 65 times for the RMAT networks and 28 times for SSCA#2 networks. When the number of the vertices is set at 32,768, the speedup rate is 17.5 for RMAT networks with the degree 10 and 46.6 for the degree 100. For SSCA#2 networks, the speedup rate is 22.1 for the degree 10 and 45.7 for the degree 100.



	10	20	30	40	50	60	70	80	90	100
GPU-rmat	10.6	15.3	19.7	24.8	29.8	30.8	32.9	37.1	40.8	44.6
GPU-ssca2	10.7	15.8	20.1	24.9	29.8	36.3	40.2	45.7	51.4	55.5
CPU-rmat	185	372	559	768	952	1154	1376	1612	1826	2082
CPU-ssca2	236	471	700	952	1177	1468	1678	1905	2240	2537

Figure 29: The bar graph and table of the results for the R-MAT and SSCA2 networks with a fixed numbers of vertices of 32,768. The computation times do not include the process of calculating the structural similarity.

When the number of vertices is fixed to the 32,768 as shown in Figure 29, the speedup ratio is approximately 17 times for the degree of 10 and it is almost 47 times for the average degree of 100. Table 2 shows all the speedup rates for RMAT and SSCA2 networks when the pre-computed similarity values are used by our algorithm.

Average Degree 6			32,768 Nodes		
Nodes	**RMAT**	**SSCA#2**	**Degree**	**RMAT**	**SSCA#2**
1024	2.2	1.9	10	17.5	22.1
2048	4.5	3.5	20	24.3	29.7
4096	8.7	5.3	30	28.4	34.8
8192	9.0	5.4	40	30.9	38.2
16384	13.3	7.9	50	32.0	39.5
32768	17.3	9.7	60	37.5	40.4
65536	24.4	16.0	70	41.8	41.7
131072	37.2	16.8	80	43.5	41.7
262144	49.6	24.1	90	44.8	43.6
524288	64.6	28.0	100	46.6	45.7

Table 2: Speedup rates with pre-computed similarity for the graphs with a constant average node degree (Left), and for the graphs with a constant number of nodes but varying average node degree (Right).

Table 2 shows that even though we use pre-computed structural similarities, our algorithm is still significantly faster than SCAN. Since we could not test SCAN for much larger networks, we only have outcomes for the networks with 524288 nodes. However, the increase in the speedup rates proves that as the size of networks increase we get much better results comparing to SCAN.

Chapter 6: Conclusion

In this study, we have introduced a *parallel structural network clustering* algorithm which is a GPU-based parallel version for SCAN network clustering algorithm. We have outlined the tasks of redesigning and parallelizing an optimized sequential algorithm designed for execution on the CPU into a massively parallel algorithm greatly accelerated by the GPU using C for CUDA. The performance of the GPU accelerated structural clustering can be more than 250 times faster than a CPU implementation. The results also indicate that this speedup becomes greater for networks with larger numbers of edges. Moreover, the parallel version we implemented is scalable it works for very large networks if the number of GPUs is increased.

Our implementation is generic enough to be executed on any CUDA enabled GPU with compute capability of 2.0 or greater. With small changes to the parameters of the kernel calls, GPUs with at least 1GB of device memory can structurally cluster networks several times larger than those documented in the performance analysis of this thesis. However, the current implementation requires that for each step the necessary data structures exist entirely in device memory. Hence, our parallel version is limited by the amount of device memory available on the GPU. However, if multiple GPUs are included in the system this problem will be solved since our algorithm is scalable.

References

[1] P. F. Drucker, "The post-capitalist society," *HarperBusiness,* 1993.

[2] R. G. Hariis, "The knowledge-based economy: intellectual origins and new economic perspectives," *International Journal of Management Reviews,* vol. 3, pp. 21-40, 2001.

[3] S. Wasserman and K. Faust, Social network analysis, Cambridge, MA: Cambridge University Press, 1994.

[4] P. J. Hinds and S. Kiesler, "Communication across boundaries: Work, structure, and use of communication technologies in a large organization.," *Organization Science,* vol. 6 (4), pp. 373-393, 1995.

[5] E. Lazega, The Collegial Phenomenon: The Social Mechanisms of Cooperation Among Peers in a Corporate Law Partnership., Oxford University Press: Oxford, UK, 2001.

[6] D. J. Watts, "The "new" science of networks," *Annual Review of Sociology,* vol. 30, pp. 243-270, 2004.

[7] B. Bollobas, Modern Graph Theory, New York, NY: Springer, 1998.

[8] J. Leskovec, J. Kleinberg and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations.," *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining..*

[9] L. C. Freeman, The Development of Social Network Analysis: A Study in the Sociology of Science, North Charleston, SC: BookSurge, 2004.

[10] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review,* vol. 45, pp. 167-256, 2003.

[11] C. Ding, X. He, H. Zha, M. Gu and H. Simon, "A min-max cut algorithm for graph partitioning and data clustering," *Proc. of ICDM,* 2001.

[12] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," *Phys. Rev. E 69, 066133,* 2004.

[13] W. W. Zachary, "An information flow model for conflict and fission in small groups," *Journal of Anthropological Research 33,* pp. 452-473, 1977.

[14] Science Direct, [Online]. Available:

http://www.sciencedirect.com/science/article/pii/S0167865509003043. [Accessed 2012].

[15] B. Bisci, Network design basics for cabling professionals., McGraw-Hill Professional.

[16] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 22, 2000.

[17] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. ,* 2004.

[18] M. E. J. Newman, "Modularity and community structure in networks," *Proc. Natl. Acad. Sci. USA in press,* 2006.

[19] J. Duch and A. Arenas, "Community detection in complex networks using extremal optimization," *Physical Review,* vol. 72, 2005.

[20] A. Clauset, " Finding local community structure in networks," *Physical Review E,* vol. 72, 2005.

[21] A. Clauset, M. E. J. Newman and C. Moore, "Finding community in very large networks," *Physical Review E 70,* 2004.

[22] R. Guimera and L. A. N. Amaral, "Functional cartography of complex metabolic networks," *Nature,* vol. 433, pp. 895-900, 2005.

[23] M. E. J. Newman, "Finding community structure in networks using eigenvectors of matrices," *Phys. Rev. E 74, 036104,* 2006.

[24] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proc. Natl. Acad. Sci. USA 99,* pp. 7821-7826, 2002.

[25] M. E. J. Newman, Networks: an introduction, Oxford, UK: Oxford University Press, 2010.

[26] P. Pons and M. Latapy, "Computing Communities in Large Networks Using Random Walks," *ISCIS,* pp. 284-293, 2005.

[27] G. K. Orman and V. Labatut, "A comparison of community detection algorithms on artificial networks," *J. Gama et al., LNAI 5808 Berlin Heidelberg, Springer-Verlag,* p. 242–256, 2009.

[28] A. Lancichinetti, S. Fortunato and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Phys. Rev. E Stat. Nonlin Soft. Matter Phys. 78, 46110 ,* 2008.

[29] M. Ester, H. P. Kriegel, J. Sander and X. Xu, "A density-based algorithm for discovering

clusters in large spatial databases with noise," *In Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD'96) ,* pp. 291-316, 1996.

[30] X. Xu, N. Yuruk, Z. Feng and T. A. J. Schweiger, "SCAN: Structural clustering algorithm for networks," *Proc. of SIGKDD,* pp. 824-833, 2007.

[31] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature,* vol. 393, pp. 440-442, 1998.

[32] J. Dongarra and T. Haigh, "Biographies," *IEEE Annals of the History of Computing,* vol. 30, pp. 74-81, 2008.

[33] A. Grama, G. Karypsis, A. Gupta and V. Kumar, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison-Wesley, 2003.

[34] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank and C. Young, "Millisecond-scale molecular dynamics simulations on anton," *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis,* pp. 1-11, 2009.

[35] R. Duncan, "A survey of parallel computer architectures," *Computer,* vol. 23, pp. 5-16, 1990.

[36] J. L. Hennessy and D. A. Patterson, Computer Architecture, Fourth Edition: A Quantitative Approach, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[37] L. R. Scott, T. Clark and B. Bagheri, Scientific Parallel Computing, Princeton, NJ, USA: Princeton University Press, 2005.

[38] J. Feo, D. Harper, S. Kahan and P. Konecny, "Eldorado," *CF '05: Proceedings of the 2nd conference on Computing frontiers,* pp. 28-34, 2005.

[39] P. Kongetira, K. Aingaran and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro,* vol. 25, pp. 21-29, 2005.

[40] "Other Message PAssing Systems," [Online]. Available: http://www.netlib.org/utk/lsi/pcwLSI/text/node70.html.

[41] M. Forum, "MPI: A Message-Passing Interface Standard. Version 2.2," [Online]. Available: http://www.mpi-forum.org. [Accessed March 2012].

[42] OpenMP, "OpenMP Application Program Interface," [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf. [Accessed 2012].

[43] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro,* vol. 30(2), 2010.

[44] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns and V. S. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *J Comput. Chem.,* 2009.

[45] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," *In HiPC,* 2007.

[46] NVIDIA, "CUDA programming guide," [Online]. Available: http://developer.download.nvidia.com/compute/cuda.

[47] NVIDIA, "Cuda Thrust Library," [Online]. Available: www.nvidia.com/content/GTC2010/pdfs/2219_GTC2010.pdf.

[48] K. A. Hawick, A. Leist and D. P. Playne, "Parallel graph component labelling with GPUs and CUDA," *In Parallel Computing 36,* pp. 655-679, 2010.

[49] R. Niewiadomski, J. Amaral and R. Holte, "A parallel external-memory frontier breadth-first traversal algorithm for clusters of work-stations," *In IEEE ICPP,* 2006.

[50] J. Soman, K. Kothapalli and P. J. Narayanan, "Some GPU algorithms for graph connected components and spanning tree," *Parallel Processing Letters 20(4),* pp. 325-339, 2010.

[51] K. Wu, E. Otoo and K. Suzuki, "Optimizing two-pass connected-component labeling algorithms," *Pattern Analysis Applications 12(2) 117135,* 2009.

[52] D. S. Hirschberg, A. K. Chandra and D. V. Sarwate, "Computing connected components in parallel computers," *Communications of the ACM 22, 8,* pp. 461-464, 1979.

[53] O. Kalentev, A. Rai, S. Kemmitz and R. Schneider, "Connected component labeling on a 2D grid using CUDA".

[54] V. Vineet, P. Harish, S. Patidar and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the GPU," *In HPG 09: Proceedings of the Conference on High Performance Graphics,* pp. 167-171, 2009.

[55] D. A. Bader and K. Madduri, "GTgraph: A synthetic graph generator suite," 2006. [Online]. Available: http://www.cc.gatech.edu/kamesh/gtgraph.