GPUSCAN: GPU-Based Parallel Structural Clustering Algorithm for Networks

Thomas Ryan Stovall, Sinan Kockara, *Member, IEEE*, and Recep Avci

Abstract—This paper presents a massively parallel implementation of a prominent network clustering algorithm, the structural clustering algorithm for networks (SCAN), on a graphical processing unit (GPU). SCAN is a fast and efficient clustering technique for finding hidden communities and isolating hubs/outliers within a network. However, for very large networks, it still takes considerable amount of time. With the introduction of massively parallel Compute Unified Device Architecture (CUDA) by Nvidia, applications properly employing GPUs are demonstrating high speed up. In current study, GPUSCAN, a CUDA based parallel implementation of SCAN, is presented. SCAN's computation steps have been carefully redesigned to run very efficiently on the GPU by transforming SCAN into a series of highly regular and independent concurrent operations. All intermediate data structures are created in the GPU to efficiently benefit from GPU's memory hierarchy. How these structures reformed and represented in the GPU memory hierarchy are illustrated. Now, through GPUSCAN, a large network or a batch of disjoint networks can be offloaded to the GPU for very fast and equivalent structural clustering. The performance of the GPU accelerated structural clustering has been shown to be much faster than the sequential CPU implementation. Both GPUSCAN and SCAN are tested on different size artificial and real-world networks. Results indicate that network becomes larger GPUSCAN significantly over performs SCAN. In tested datasets, speed-up of over 500-fold is achieved. For instance, calculating structural similarity and clustering of 5.5 million edges of the California road network in GPUSCAN is 513-fold faster than the serial version of SCAN.

Index Terms—Structural clustering for networks, GPGPU, high-performance computing, very large networks, social networks

1 Introduction

Networks offer a powerful analogy for modeling relational data in areas ranging from bioinformatics [1] to social network analysis [2]. Consider the network given in Fig. 1. For instance in a social network domain; the nodes in the network may represent a set of people, while each edge (connection) represents any relationship between two people.

We are living in an era of very large networks with millions of nodes and connections. For instance, as of September 30, 2013, Facebook has over 874 million monthly users [28]. Automated network analysis techniques can provide insights of relations among the nodes that can lead to identify key people, documents, events, etc. These automated techniques can elucidate connections otherwise remain unnoticeable. The massiveness of current networks necessitates development of massively parallel algorithms for network analysis. There are numerous parallel applications and algorithms for analyzing graphs. Among those are GraphCT [29], the parallel boost graph library (PBGL) [30], Google's Pregel [31], parallel semantic graph analysis [32], graph partitioning for distributed graph computations [33], and open-source scalable graph analysis toolbox [34] to name a few.

• *The authors are with the Department of Computer Science at University of Central Arkansas, Conway, AR 72035, USA.*
E-mail: {trstovall, ravci}@uca.edu, skockara@gmail.com.

Manuscript received 29 Jan. 2014; revised 17 Oct. 2014; accepted 11 Nov. 2014. Date of publication 25 Nov. 2014; date of current version 6 Nov. 2015.
Recommended for acceptance by A. Mellouk.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2374607

Basic community detection 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. These include min-max cut [3], normalized cut [4], various modularity based algorithms [1], [2], [5], the SCAN [6], and SHRINK [25]. Graph clustering is grouping of vertices that are more strongly linked to each other than other vertices. Especially for very large networks finding and grouping these strong connections are very important in different domains. For instance, in the context of sociology this may correspond to community detection in social networks such as Twitter, or detecting emerging research topics in citation networks. In the context of disease network, graph clustering may correspond to an early detection of an epidemic. In the context of preventing criminal activity, graph clustering may correspond to finding network of terrorist cells from their mobile or internet communications.

In 2007, SCAN, a network clustering algorithm that is a simple adaption of a prominent density-based data clustering approach DBSCAN [7], was introduced as an alternative to the Girvan-Newman based modularity algorithms [2]. Rather than using betweenness [8] 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-graphs which represent the hidden communities within the network. Some nodes do not share enough relationships with any particular cluster. Thus, they do not merit being assigned to a cluster. If such a node

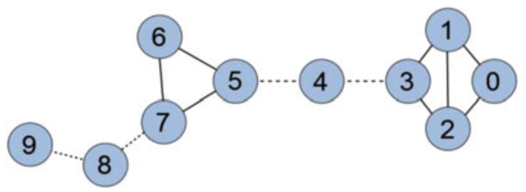

Fig. 1. A network with two Clusters {0, 1, 2, 3}, {5, 6, 7}, a Hub {4}, and two Outliers {8, 9}.

bridges two or more clusters, it is then classified as a hub [9]. Otherwise, it is classified as an outlier and may be regarded as a noise.

Considering the rapidly evolving technology of graphics processing units (GPUs), these powerful co-processors have improved rendering of 3D environments for games and 3D CAD software for decades. Scientific computing has also shared in the benefits of high instruction throughput, but required using obscure graphics APIs. With the release of Nvidia's CUDA enabled graphics cards, general purpose computing can now be performed on consumer GPUs using an extension to standard C++ [10].

In this study, we redesigned and parallelized SCAN as a series of highly regular and independent operations in order to optimally 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. Following sections provide a brief review of SCAN, and discuss the computation steps of GPU-based parallel structural clustering algorithm for networks (GPUSCAN). Later results are presented.

1.1 General Purpose Computing in GPU As a Coprocessor

Coprocessing units have long been used to supplement the functionality of the central processing unit (CPU). Math coprocessors were introduced in the 1970s to add scientific computing capabilities to word processing computers. Today, GPUs provide the most computational power for the price. With current graphics cards, scientific applications are able to harness that computational power. In the last few years, CUDA has accelerated many non-graphical applications, with up to a few hundred times speedup over sequential CPU execution [11], [12], [13].

Massively parallel GPUs can accelerate general purpose computing; however, mapping computation of any algorithm 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. The most difficult challenge is often utilizing high bandwidth memory in the GPU and managing this hierarchical memory. Parallel GPU implementation of any algorithm is still very implementation dependent. Inefficient use of special memories in GPUs is major limiting factors of performance. Therefore, parallel GPU implementation of SCAN required total redesign of the algorithm in order to optimally benefit from available memories in the GPU hardware. A CUDA graphics card is composed of a set of multiprocessors and each multi-processor composed of a set of streaming processors called CUDA cores. Each multiprocessor 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. So, coalesced memory access (retrieving an entire block from memory) is strongly encouraged.

1.2 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. The thread block consists of a set of warps, where a thread warp is a set of 32 threads executing in single instruction multiple data (SIMD) parallel. 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.

Our implementation of GPUSCAN uses the Thrust library [14] since procedures such as sorting; greatly increase coalesced memory access. The Thrust template library is a parallel implementation of the C++ Standard Template Library for CUDA enabled GPUs. Sorting in GPUSCAN is performed using radix sort from the Thrust library.

2 SCAN

SCAN uses the notion of structural similarity to agglomerate nodes of a network into clusters [6]. The algorithm needs two parameters. The first, ε, is the threshold structural similarity value for adjacent nodes to be considered epsilon-neighbors (ε-neighbor, N_{ε}). The second parameter, μ, is a threshold for the minimum number of ε-neighbors that a node must have to be considered a core node (4). μ is set to 2 in expression (4). The vertex structure (1) of an arbitrary vertex u, $\Gamma(\mathrm{u})$, from a graph is given as the set of u and the vertices adjacent to u. Two vertices u and v have a structural similarity $\sigma(\mathrm{u,v})$ (2) based on the number of nodes common to the vertex structures of both nodes. For SCAN, each node in a cluster must be a core node or an ε-neighbor of a core node. Algorithm 1 summarizes steps of the SCAN algorithm with a small optimization which will be explained below.

$$\Gamma(\mathrm{u}) = \{\mathrm{v} : (\mathrm{u},\mathrm{v})\,\varepsilon\,\mathrm{E}\} \cup \{\mathrm{u}\} \tag{1}$$

$$\sigma(\mathrm{u},\mathrm{v}) = \frac{|\Gamma(\mathrm{u}) \cap \Gamma(\mathrm{v})|}{\sqrt{|\Gamma(\mathrm{u})| * |\Gamma(\mathrm{v})|}} \tag{2}$$

$$\mathrm{N}_{\varepsilon}(\mathrm{u}) = \{\mathrm{v} : \sigma(\mathrm{u},\mathrm{v}) \geq \varepsilon\} \tag{3}$$

$$\mathrm{CORE}_{\varepsilon,\mu}(\mathrm{u}) \leftrightarrow |\mathrm{N}_{\varepsilon}(\mathrm{u})| \geq \mu. \tag{4}$$

Algorithm 1. Optimized Serial Version of SCAN

```
optimized serial SCAN
input:
  G = {V,E}, epsilon (ε), mu
  Q is queue
  u, t, v are vertices

procedure SCAN(G = {V,E}, epsilon, mu):
cluster_id = 0
for each v in V:
  if v.cluster is null
    enqueue v onto Q
    while Q is not empty:
      t ← dequeue(Q)
      if isCore(t, epsilon, mu):
        // first visit to this vertex
        if t.cluster is null:
          cluster_id + = 1
          t.cluster ← cluster_id
        end if
        if t.cluster is cluster_id:
          for each u in t.epsilon_neighbors:
            // only visit each node once
            if u.cluster is null:
              u.cluster ← cluster_id
              enqueue u on Q
            // is candidate for multiple clusters
            else if u.cluster is not cluster_id:
              u.cluster ← HUB
            end if //if u.cluster is null
          end for
        end if //if t.cluster is cluster_id
      else if t.cluster is null //if isCore(t, epsilon, mu)
        t.cluster ← OUTLIER
      end if // if isCore(t,epsilon,mu)
    end while
  end if //if v.cluster is null
end for

procedure isCore(u, epsilon, mu):
for each v in u.adjacencyList:
  if v in u.epsilon_neighbors:
    continue // similarity has already been calculated
  else if (v.cluster is not null) and u.is_core and v.is_core:
    continue // similarity does not need to be calculated
  else if similarity(u, v)>= epsilon:
    insert u into v.epsilon_neighbors
    insert v into u.epsilon_neighbors
    if length(v.epsilon_neighbors)>= mu:
      v.is_core = True
    end if
    if length(u.epsilon_neighbors)>= mu:
      u.is_core = True
    end if
  end if
end for
return u.is_core
```

SCAN initially labels all vertices as unclassified. Then a vertex v is arbitrarily chosen from the vertex list V. Structural similarity is calculated for each edge. However, in Algorithm 1, some of the edges' structural similarity are not calculated to optimize the performance. In SCAN, two vertices are epsilon neighbors if they have a structural similarity of at least epsilon. Furthermore, if a vertex v is an epsilon neighbor of some vertex u and either u or v have at least mu epsilon-neighbors, then u and v may be in the same cluster. Thus, if u and v are known to be in the same cluster, then computing structural similarity of the two vertices is redundant. Optimized serial SCAN given in Algorithm 1 avoids this redundant computation by comparing cluster identifiers. If two vertices are core vertices and either both are in the same cluster or one is a hub vertex, then structural similarity is not computed.

Hence, optimized serial SCAN initializes all vertex labels to unclassified. For each arbitrarily chosen vertex v in vertex list V, if v is unclassified, then v is enqueued into a new queue Q. Then, while Q is not empty, for each vertex t dequeued from Q, the structural similarity to t is calculated for each vertex in the adjacency list of t (t.epsilon_neighbors). If t has at least mu epsilon neighbors, then all unclassified epsilon neighbors of t are enqueued onto Q (enqueu u onto Q). If t has no cluster identifier, then a unique cluster identifier is generated. The cluster identifier is assigned to t and all unclassified epsilon neighbors of t as the vertex classification. Each epsilon-neighbor of t that is classified with a cluster identifier different from t's cluster identifier is reclassified as a HUB vertex. Otherwise, if t has less than mu epsilon neighbors (t.cluster is null), then it is classified as an OUTLIER. In the best case scenario, this optimization will reduce approximately 33 percent of structural similarity computations. As seen from Algorithm 1, some nodes are not assigned in any cluster. These unclassified non-members in later steps are further classified into hubs and outliers in Algorithm 1 i.e. if a non-member node is a core node but do not have a cluster, it is considered a hub. Otherwise it is an outlier.

Computation of SCAN for very large networks takes a large amount of time. Many set intersection operations must be performed. To reduce the computation time, we perform the necessary computations on the GPU using the massively multithreaded CUDA architecture. Redesigned structural clustering algorithm is explained in the following section.

3 GPUSCAN

The goal of constructing any GPU based algorithm is to offload data parallel and computationally intensive pieces of the algorithm in to the GPU. For our GPU based parallel implementation of SCAN (GPUSCAN), the network is copied in to the GPU memory as a pair of integer arrays representing edges of the network. On the device (GPU) a series of computations are performed, leaving the CPU only necessary for global thread synchronization. Once everything is computed on the device, the result is an integer array describing the cluster membership (labels) of each vertex.

Algorithm 2 illustrates the general work flow of GPUSCAN. The algorithm consists of three main parts: identifying ε-neighbor pairs, computing connected components to find clusters, and classifying nodes (hubs or outliers) which currently do not belong to any cluster. GPUSCAN begins

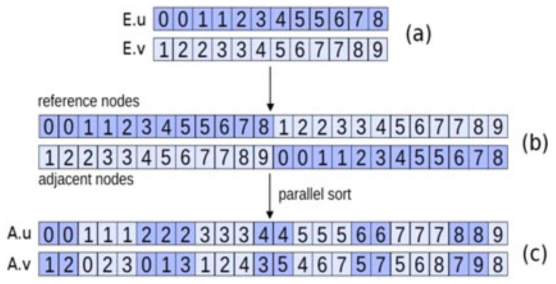
E.u 0 0 1 1 2 3 4 5 5 6 7 8
E.v 1 2 2 3 3 4 5 6 7 7 8 9
(a)
reference nodes
0 0 1 1 2 3 4 5 5 6 7 8 1 2 2 3 3 4 5 6 7 7 8 9
1 2 2 3 3 4 5 6 7 7 8 9 0 0 1 1 2 3 4 5 5 6 7 8
(b)
adjacent nodes
parallel sort
A.u 0 0 1 1 1 2 2 2 3 3 3 4 4 5 5 5 6 6 7 7 7 8 8 9
A.v 1 2 0 2 3 0 1 3 1 2 4 3 5 4 6 7 5 7 5 6 8 7 9 8
(c)

Fig. 2. The set of adjacency list A (c) is derived directly from the edge list E (a) through concatenating E.u, E.v (b) then sorted (c).

process by copying the edge list E, represented as a pair of integer arrays, from CPU (host) memory in to the device memory (E.u and E.v). As illustrated in Fig. 2, to create a set of adjacency lists, E.u is copied in to the first half of A.u and the second half of A.v. Then, E.v is copied in to the second half of A.u and the first half of A.v. The arc list A (Fig. 2b) is represented by A.u and A.v on the device. Then they sorted lexicographically using the parallel radix sort functions from the Thrust library, see Fig. 2c.

Algorithm 2. GPUSCAN

```
host function GPUSCAN($G(V, E)$, $\varepsilon$ , $\mu$)
input: $E$ edge list of graph $G := (V, E)$. $V$ is the sequence $0..(|V| - 1)$ representing the vertices in $G$. $\varepsilon$ is the thresh old structural similarity for neighbors. $\mu$ is threshold number of neighbors for $CORE$ vertex.
output: $labels$ gives the cluster label or outlier class.
declare integer $labels[|V|]$ in host memory
declare integer $E.u[2|E|]$, $E.v[2|E|]$ in device
declare integer $A.u[2|E|]$, $A.v[2|E|]$ in device
declare boolean $is\ core[|V|]$ in device
declare integer $parent[|V|]$ in device
  $E.u \leftarrow E[0]$ // host to device
  $E.v \leftarrow E[1]$ // host to device
  $A.u \leftarrow E.u$ // device to device
  $A.v \leftarrow E.v$ // device to device
  $A.u + |E| \leftarrow E.v$ // device to device
  $A.v + |E| \leftarrow E.u$ // device to device
call sort by key($A.v$, $A.v + 2|E|$, $A.u$)
// using Thrust library for sorting
call sort by key($A.u$,$A.u+2|E|$,$A.v$)
call ε_Neighbors($\varepsilon$ , $E.u$, $E.v$, $A.u$, $A.v$)
call coreVertices($\mu$, $E.u$, $E.v$, $is\ core$)
call linking($E.u$, $E.v$, $parent$)
call contraction($parent$)
call idClusters($is\ core$, $parent$)
call classNonMembers($A.u$, $A.v$, $parent$)
  $labels \leftarrow parent$ // device to host
```

3.1 Identifying ε-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 Fig. 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}. Fig. 3 illustrates the data structures necessary for calculating these set intersections in the GPU.

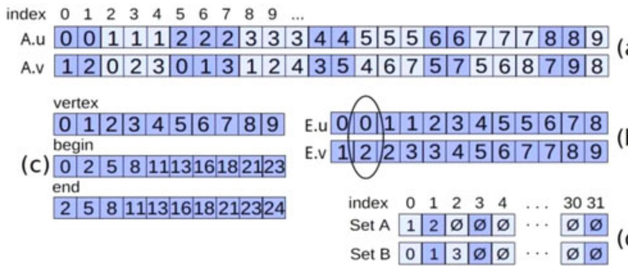
index 0 1 2 3 4 5 6 7 8 9 ...
A.u 0 0 1 1 1 2 2 2 3 3 3 4 4 5 5 5 6 6 7 7 7 8 8 9
A.v 1 2 0 2 3 0 1 3 1 2 4 3 5 4 6 7 5 7 5 6 8 7 9 8
(a)
vertex
0 1 2 3 4 5 6 7 8 9
begin
0 2 5 8 11 13 16 18 21 23
end
2 5 8 11 13 16 18 21 23 24
(c)
E.u 0 0 1 1 2 3 4 5 5 6 7 8
E.v 1 2 2 3 3 4 5 6 7 7 8 9
(b)
index 0 1 2 3 4 . . . 30 31
Set A 1 2 Ø Ø Ø · · · Ø Ø
Set B 0 1 3 Ø Ø · · · Ø Ø
(d)

Fig 3. Data structures used for calculating structural similarity in GPU. The set of adjacency lists (a) is indexed (c). Then for each edge (e.g. (0,2)) (b) the adjacency list of the first node is compared to the adjacency list of the second node (d), the number of matches are counted, then structural similarity for the edge is calculated according to Equation (6).

Algorithm 3. Finding Epsilon Neighbors

```
host function ε _Neighbors($\varepsilon$, $E.u$, $E.v$, $A.u$, $A.v$)
input: $\varepsilon$ is the threshold structural similarity for $\varepsilon$ neighbors. $E.u$ and $E.v$ form the edge list $E$ in device mmemory. $A.u$ and $A.v$ form the list of adjacent pairs $A$.
output: $E.u$ and $E.v$ modified to form the list of epsi lon neighbor pairs $N$

declare integer $begin[|V|]$, $end[|V|]$ in device
declare boolean $flag[|E|]$ in device
using $2^* |E|$ parallel threads on the device
execute:
  ID_BOUNDARIES($A.u$, $begin$, $end$)
declare integer $numWarps \leftarrow$ floor(($|E| + 31)/32$)
using 32 $^*$ $numWarps$ parallel threads on the device
execute:
  ID_ε_NEIGHBORS($\varepsilon$,$E.u$, $E.v$, $A.v$, $begin$, $end$, $flag$)
declare integer $N.size$
$N.size \leftarrow$call partition($\{E.u, E.v\}$, \//$cont. next line$
                    $\{E.u, E.v\} + |E|, E.u = null$)
  $E.u + N.size \leftarrow E.v$ // device to device
  $E.v + N.size \leftarrow E.u$ // device to device
resize $E.u$ and $E.v$ to 2 $^*$ $N.size$
```

The adjacency list for each node is extracted from the arc list A. A.u gives the reference nodes and the second array A. v gives the adjacent nodes. The first and last occurrence of each node in A is recorded as starting and ending index positions (Fig. 3c). The indices are stored in the integer arrays begin and end by the device kernel ID_BOUNDARIES (See Algorithm 4) and used to calculate structural similarities. This indexing is pleasingly parallel and uses coalesced memory accesses. For instance, the GTX 480 (having 512 CUDA cores) can test 512 elements in A.u at a time. The following formulations show how GPUSCAN calculates the structural similarity for two nodes u and v (See Algorithm 3):

$$\mathrm{adj(u)} = \{\mathrm{v} : (\mathrm{u}, \mathrm{v}) \in \mathrm{A}\} \tag{5}$$

$$\sigma(\mathrm{u}, \mathrm{v}) = \frac{2 + |\mathrm{adj(u)} \cap \mathrm{adj(v)}|}{\sqrt{(1 + |\mathrm{adj(u)}|)*(1 + |\mathrm{adj(v)}|)}}, \tag{6}$$

where adj(u) is the adjacency list of the node u.

Algorithm 4. Identifying vertex boundaries in sorted edge lists

```
device kernel ID BOUNDARIES(A.u, begin, end)
declare integer smem[] in shared memory
declare integer tid ← global thread ID
declare integer bid ← tid mod blocksize
  smem[bid] ← A.u[tid]
if smem[bid] ! = smem[bid – 1] then
  begin[smem[bid]]←tid end if
if smem[bid] ! = smem[bid + 1] then
  end[smem[bid]] ← tid + 1
end if
```

Algorithm 5. Identifying Epsilon Neighbors in Device

```
device kernel ID_ε_NEIGHBORS(ε, E.u, E.v, A.v, begin, end, flag)
declare integer tid ← global thread ID
declare integer bid ← tid mod block size
declare integer pos ← tid modulo warp size
declare integer b1[], b2[], e1[], e2[] in shared memory
declare integer setA[], setB[], count[] in shared memory
b1[bid] ← begin[E.u[tid]]
b2[bid] ← begin[E.v[tid]]
e1[bid] ← end[E.u[tid]]
 e2[bid] ← end[E.v[tid]]
for i in 0..31 do
  w pos ← bid – pos
  setA + w_pos ← A.v + b1[w pos + i]
  setB + w_pos ← A.v + b2[w pos + i]
  count[w_pos + i] ← |intersect(setA, setB)|
end for
e1[bid] ← e1[bid] – b1[bid] e2[bid] ← e2[bid] – b2[bid]
flag[tid] ← count[bid]/sqrt(e1[bid] * e2[bid]) ≥ ε
```

Now, structural similarity defined in (2) and equivalently redefined in (6) for GPUSCAN can be computed by the ID_ε_NEIGHBORS kernel (see Algorithm 5). Each warp (set of 32 threads) fetches a block of 32 integers from E.u and a block from E.v, forming a list of 32 edges in shared memory. To avoid branching, all threads in the warp compute the set intersections for each edge before advancing to the next edge. The threads in the warp work together to count the elements of intersections. For brevity, the pseudocode in ID_ε_NEIGHBORS (Algorithm 5) presumes that the adjacency list for each vertex is no longer than 32 vertices. For each edge $e = (u, v)$, the adjacency list of u is stored in setA and the adjacency list of v is stored in setB. The two sets have already been sorted, so the intersection length is counted by merge comparison. The warp makes 32 unique comparisons into one operation by comparing four elements of setA (tid/8) with eight elements from setB (tid modulo 4), counting the number of matches, and advancing either to the next four elements in setA or to the next eight elements in setB as necessary. In the implementation, if an adjacency list is longer than 32 vertices, then the contents of setA and/or setB starting at offset w_pos is replaced with the next 32 elements in the adjacency list as necessary (see Algorithm 5).

Algorithm 6. Set intersection

```
device kernel intersect(setA, setB)
input: setA[], the adjacency list of node u, setB[], the adjacency list of node v, there exists edge between u and v.
output: total, number of intersecting nodes (intersection length) of nodes u and v.
declare integer total ← 0 in shared memory
declare integer lenA ← length(setA) in shared memory
declare integer lenB ← length(setB) in shared memory
declare integer supA ← 0 in shared memory
declare integer supB ← 0 in shared memory
declare integer posA ← tid / 8 in local memory
declare integer posB ← tid modulo 8 in local memory
declare integer elemA ← 0 in local memory
declare integer elemB ← 0 in local memory
declare integer count ← 0 in local memory
while (posA < lenA) and (posB < lenB):
  elemA = setA[posA]
  elemB = setB[posB]
  if elemA = = elemB:
    count + = 1
  end if
  if (posA modulo 4 = = 3) or (lenA-1 = = posA):
    supA = elemA
  end if
  if (posB modulo 8 = = 7) or (lenB-1 = = posB):
    supB = elemB
  end if
  if supA <= supB:
    posA + = 4
  end if
  if supA >= supB:
    posB + = 8
  end if
end while
atomic_add(total, count)
return total
```

The intersect kernel receives two sorted sets from global device memory and returns the number of elements which exist in both sets. Each thread in the warp compares a unique pair of elements from the two sets in each iteration of the while loop (see Algorithm 6). Algorithm 6 includes a load operation from global memory by 32 elements of a set in a single memory fetch. The intersect kernel performs concurrent merge intersection of setA and setB returning total, the number of elements in the intersection of the two sets. The merge intersection comparisons are performed in concurrent batches of 32. Each thread with thread identifer tid compares element (elemA) at index posA from setA to element (elemB) at index posB from setB, and increments count if the two elements are equal. Note that initially posA = tid/8 and posB = tid%8, so that (posA, posB) is a unique pair for each unique tid. Then, posA is incremented by 4 if all elements from setA compared so far is less than or equal to any element compared so far from setB (if supA <= supB). Likewise, posB is incremented by 8 if all elements from setB compared so far is less than or equal to any element compared so far from setA (if supA > = supB). This corresponds directly to the sequential merge intersect algorithm, with the difference that the intersect kernel compares four elements of setA to eight elements of setB (total 32 elements) in each iteration instead of

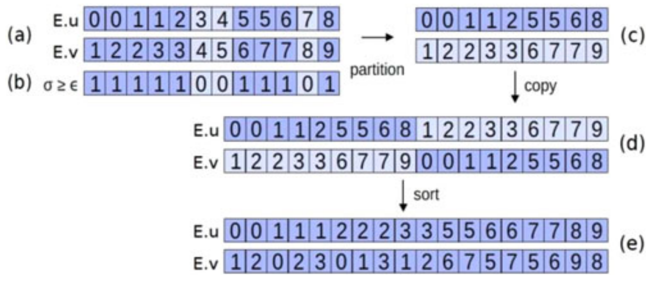
(a)
E.u 0 0 1 1 2 3 4 5 5 6 7 8
E.v 1 2 2 3 3 4 5 6 7 7 8 9
(b) σ ≥ ε 1 1 1 1 1 0 0 1 1 1 0 1
partition
0 0 1 1 2 5 5 6 8
1 2 2 3 3 6 7 7 9
(c)
copy
E.u 0 0 1 1 2 5 5 6 8 1 2 2 3 3 6 7 7 9
E.v 1 2 2 3 3 6 7 7 9 0 0 1 1 2 5 5 6 8
(d)
sort
E.u 0 0 1 1 1 2 2 2 3 3 5 5 6 6 7 7 8 9
E.v 1 2 0 2 3 0 1 3 1 2 6 7 5 7 5 6 9 8
(e)

Fig. 4. Determining the list of ε-neighbor pairs (e). The edge list (a) is pruned of edges not having a structural similarity of at least ε (c) according to the flag array (b). The remaining edges are copied (d) then sorted.

one element from each set. Finally, all threads atomically add count in local memory to total in shared memory. At the end, total is returned by the intersect kernel. The parallel set intersection kernel summarized in Algorithm 6.

Once we know which edges represent structurally similar vertex pairs, we can form the list of ε-neighbor pairs in the same way that adjacency list A was created. First, the edge list E is partitioned, moving all edges with structural similarity greater than ε (as indicated by the flag array in Algorithm 5) to the front of E.u and E.v. Supposing there are |N| many edges connecting ε-neighbor pairs, E.v is copied in to E.u at the offset of N. Similarly, E.u is copied to E.v at the offset of N. The result is the list of ε -neighbor pairs stored in E.u and E.v (see Algorithm 5). This process is illustrated in Fig. 4.

Before starting next step -clustering step-, we find which vertices are core vertices (See Equation 4). In SCAN, parameter μ controls the necessary shape and size of clusters by defining a node as having at least μ number of ε-neighbors and requiring that all nodes in a cluster be either a core node or an ε-neighbor to a core node. Hence, we identify the core nodes by counting the number of ε-neighbors for each node. This step is illustrated in Algorithm 7.

Before starting clustering step in GPUSCAN, ε-neighbor pairs which do not contain at least one core node are removed from edge list. Removal of ε-neighbor pairs from edge list is illustrated in Fig. 5. Notice that edge between nodes 8–9 is removed from edge list in this illustration. By sorting E.u and E.v lexicographically, we can use E.u to index the start and end of each node's list of ε-neighbors in E.v, executing the ID_BOUNDARIES kernel previously called by the host function ε_Neighbors. Then, determining if node k is a core node is simply is_core[tid] ← (end[k] – begin[k]) ≥ μ (See Algorithm 7). Removing all pairs from E.u and E.v which do not contain at least one core node will leave the connected components of E.u and E.v as proper clusters. Recall that the same methodology is used in optimized serial version of SCAN (see Algorithm 1).

Algorithm 7. Determining core vertices and finding their IDs in device

```
host function coreVertices(ν, E.u, E.v, is core)
input: μ, the minimum number of epsilon neighbors for a vertex to be considered a core.
output: isCore, the boolean array indicating which vertices are CORE
declare integer begin[|V|], end[|V|] in device memory
call sort by key(E.v, E.v + |E.u|, E.u)
call stable sort by key(E.u, E.u + |E.u|, E.v) using |E.u| parallel threads on the device execute:
ID_BOUNDARIES(E.u, begin, end)
using |V| parallel threads on the device execute:
begin
device kernel ID CORES(ε, begin, end, is_core)
    declare integer tid ← global thread ID
    is_core[tid] ← (end[tid] – begin[tid]) ≥ μ
end begin
```

3.2 Clustering

Clustering in SCAN is efficiently executed on the CPU as a breadth first graph traversal. However, breadth first search is irregular and inefficiently executed on the massively multi-threaded SIMD graphics cards. Fast breadth first search algorithms [26], [27] for large graphs do exist for CUDA [15], [16] but when there are many small sub-graphs to be constructed in this step, these methods become inefficient. Also, some CUDA implementations exist for connected component labeling [17], [18], [19], [20]; however, connected components in [17] can fail to converge. So, we provide a simple version for finding connected components adapted from [17] and [21]. So, GPUSCAN instead uses a two phase connected components method of turning the set of components in the network in to a set of trees (called linking), then using pointer jumping to assign a common label to nodes in the same tree (called graph contraction). These are explained in section 3.2.1 linking.

3.2.1 *Linking*

A spanning forest is a graph without cycles whereas spanning tree is a subgraph of the original graph that is fully connected spanning forest. A graph is called connected if every pair of vertices is linked. The first phase of computing structurally connected components is the linking phase which uses the sorted ε-neighbors list to assign a cluster label (parent) to nodes of a cluster. N is the list of ε neighbor pairs stored in E.u and E.v to create the integer array parent. For each node u in the network, parent stores the next node in the path from u to the root of the tree spanning the component containing u. Hence, parent holds a spanning forest for the structurally connected components formed by N. Linking begins with each node in the original graph initialized as having no parent. Then, for every iteration, each

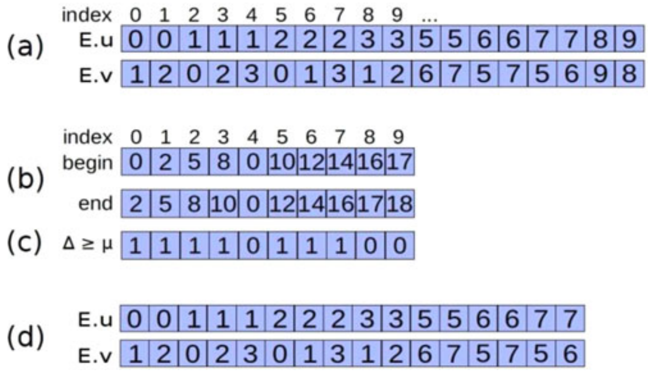
index 0 1 2 3 4 5 6 7 8 9 ...
(a)
E.u 0 0 1 1 1 2 2 2 3 3 5 5 6 6 7 7 8 9
E.v 1 2 0 2 3 0 1 3 1 2 6 7 5 7 5 6 9 8
index 0 1 2 3 4 5 6 7 8 9
(b)
begin 0 2 5 8 0 10 12 14 16 17
end 2 5 8 10 0 12 14 16 17 18
(c) Δ ≥ μ 1 1 1 1 0 1 1 1 0 0
(d)
E.u 0 0 1 1 1 2 2 2 3 3 5 5 6 6 7 7
E.v 1 2 0 2 3 0 1 3 1 2 6 7 5 7 5 6

Fig. 5. ε-neighbor pairs (a) which do not contain at least one core node are removed. The list of pairs is indexed (b), and the difference, begin – end, is compared to μ. After removing pairs which do not contain a core node (d), connected components identified in E are proper clusters.

node is assigned to a parent from the set of itself and its ε-neighbors. On odd iterations, the minimum node identifier in the set is chosen as the parent (including itself). On even iterations, the maximum node identifier is chosen as the parent. Alternating between minimum and maximum (odd and even) is an optimization and tends to select a central node for the root of the tree, thereby allowing for a shorter tree. At the end of the each iteration, the nodes in N (epsilon neighbor pairs, E.u[] and E.v[]) are replaced by their parents. Pairs of identical node identifiers (self-links) are removed from the N, and then N is sorted. Linking terminates once N is empty, resulting in a spanning forest of the structurally connected components stored in a parent. This is illustrated in Fig. 6 and summarized in Algorithm 8.

Algorithm 8. Linking Clusters

```
host function **linking**(E.u, E.v, parent)
**input:** E.u, E.v representing the neighbors list N
**output:** parent, the array of component labels
using |V| parallel threads on the device do:
  **initialize** parent[u] ← u $\forall$ u $\in$ V
  **declare** boolean odd ← false
**while** |E.u| ! = 0 **do**
  odd ← !odd // alternate odd/even
// zip, sort, and partition are in the Thrust CUDA library. zip:
//treat a structure of arrays as an array of structures
  E ← call zip(E.u, E.u + |E.u|, E.v)
// sort: sort array of structures (x,y) by x, then y
// sort ascending on odd iterations, descending on even
  **call** sort(E, E + |E|, ascending = odd)
  using |E.u| parallel threads on the device execute:
**begin**
  device kernel LINKING(E.u, E.v, parent, odd)
    **declare** integer tid ← global thread ID
    **declare** integer bid ← tid modulo block size
    **declare** boolean do_copy ← False
    **declare** integer u[] in shared memory
    u[bid] ← E.u[tid]
    **if** tid = 0 **then** // first thread in kernel
      do_copy ← True
    **else if** bid = 0 **then** // first thread in block
      **if** u[0] ! = E.u[tid−1] **then** // start of adj list for u[tid]
        do_copy ← True
      **end if**
  **else if** u[bid] ! = u[bid−1] **then** // start of adj list for u[bid]
        do_copy ← True
  **end if**
  **if** do_copy **then**
  parent[u[bid]] ← odd ? min(u[bid], E.v[tid]) :
                                max(u[bid], E.v[tid])
  **end if**
**end begin**
using |E.u| parallel threads on the device execute:
**begin**
  device kernel REMOVE SELF LINKS(E.u, E.v,parent)
    **declare** integer tid ← global thread ID
    **declare** integer u ← parent[E.u[tid]]
    **declare** integer v ← parent[E.v[tid]]
    **if** u = v **then**
      E.u[tid] ← null
    **else**
      E.u[tid] ← u, E.v[tid] ← v
    **end if**
  **end begin**
// zip: treat a structure of arrays as an array of //structures
  E ← call zip(E.u, E.u + |E.u|, E.v)
// partition: move elements to beginning where key
//evaluates to true. It returns the number of elements //of E.u
which are not null
 |E.u| ← **call** partition(E, E + |E|, key = lambda e: e[0]! = null)
**end while**
```

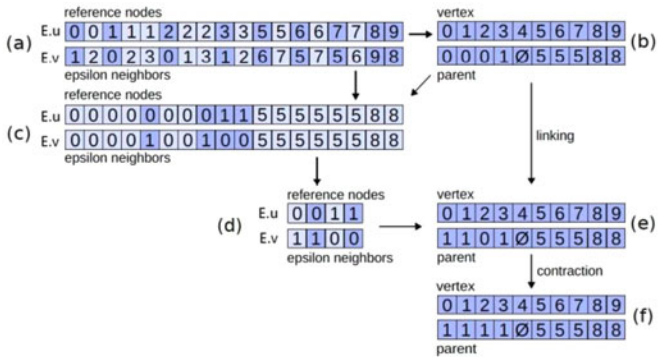

Fig. 6. Connected components as implemented in GPUSCAN.

In Algorithm 8, the variable 'odd' alternates between 'true' and 'false'. On the first iteration, 'odd' is true, so the algorithm looks for the minimum value of v to declare as the parent of node u. Hence, for each node u[i], v[i] is a candidate parent of u[i] if u[i-1] is not u[i]. The parent is actually chosen to be minimum(u[i], v[i]), since the existence of (u,v) implies (v,u) is also in the set. Such is the case for nodes 0, 5, and 8 in Fig. 6. For instance, for node 5 among its epsilon neighbors, minimum valued epsilon neighbor is 6. However, node 6's minimum valued epsilon neighbor is 5. Since edge between (5), (6) also implies existence of (6), (5), minimum(5), (6) results 5. Thus, node 5 is selected as a parent for both nodes 5 and 6.

Parent of a node in Fig. 6b called 'linking'. Nodes in the list of ε-neighbor pairs are replaced by their parents in Fig. 6c and self-links are removed in Fig. 6d. In the next iteration, the highest neighbor id (in dark blue) is chosen as the new parent as seen in Fig. 6e. Again, nodes in the pairs list are replaced by the parents and also self-links are removed. Pointer jumping or "graph contraction" on the parent list in Fig. 6e produces the cluster labels for the nodes belonging to final clusters as seen in Fig. 6f.

3.2.2 Graph Contraction

Once a component is connected by a spanning tree, partitioning the set of nodes into connected components is trivial. Labeling is accomplished using the method of pointer doubling where in each iteration until no change is made; the parent p of a node is replaced by the parent of p, for all nodes where these nodes' parent is p (changed to parent of a parent). For global thread synchronization, each iteration is handled by a new kernel call. Graph contraction concludes with a disjoint set of structurally connected components which are proper clusters. The label given to each member of a cluster is the node identifier of the root of the tree spanning that cluster. These steps are summarized in Algorithm 9 and illustrated in Fig. 6e and 6f.

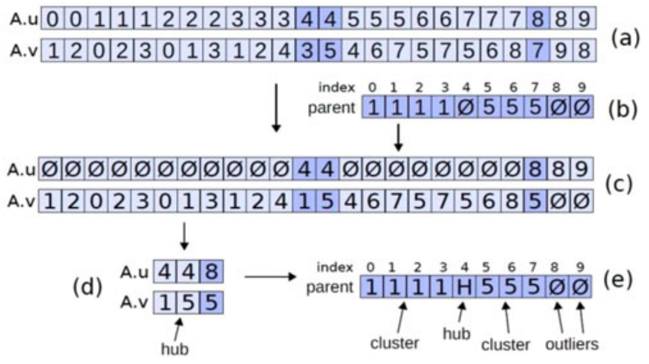
A.u 0 0 1 1 1 2 2 2 3 3 3 4 4 5 5 5 6 6 7 7 7 8 8 9
A.v 1 2 0 2 3 0 1 3 1 2 4 3 5 4 6 7 5 7 5 6 8 7 9 8
(a)
index 0 1 2 3 4 5 6 7 8 9
parent 1 1 1 1 Ø 5 5 5 Ø Ø
(b)
A.u Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø 4 4 Ø Ø Ø Ø Ø Ø Ø Ø 8 8 9
A.v 1 2 0 2 3 0 1 3 1 2 4 1 5 4 6 7 5 7 5 6 8 5 Ø Ø
(c)
(d)
A.u 4 4 8
A.v 1 5 5
hub
index 0 1 2 3 4 5 6 7 8 9
parent 1 1 1 1 H 5 5 5 Ø Ø
(e)
cluster
hub
cluster
outliers

Fig. 7. Classifying non-member nodes as either hubs or outliers.

Algorithm 9. Graph Contraction

```
host function contraction(parent)
input: parent, the array of component labels.
output: parent, updated to connected component labels.
declare boolean p ← true
declare boolean $p_d$ in device memory
while p do
$p_d$ ← false
using |V| parallel threads on the device execute:
begin
  device kernel GRAPH CONTRACTION(parent,$p_d$)
  declare integer tid ← global thread ID
  declare integer label ← parent[tid]
  if label ! = null then
    declare integer plabel ← parent[label]
    if plabel ! = label then
      parent[tid] ← plabel
      $p_d$ ← true
    end if
  end if
end begin
  p ← $p_d$ end
end while
```

3.3 Classifying Non-Members

This step contains classification of nodes that have not grouped into any cluster yet. To do this, we use the arc list A obtained at the beginning of GPUSCAN. For each thread k, if the node at A.u[k] is a member of a cluster, then A.u [k] is set to null. Next, each node in A.v is replaced by its cluster label, where the cluster label for non-members is null. For each pair a = (u, v) ∈ A, if either u or v is null, then a is removed from A. Deletion is implemented using the parallel partition function provided by the Thrust library to move the adjacency lists of interest to the beginning of A (thereby, moving deleted items to the end). Finally, each cluster id at A.v[s] is compared to the cluster id at A[s-1]. If the two are different and the reference nodes at A.u[s] and A.u[s-1] are the same, then the node at A.u[s] is classified as a hub, as indicated by the device kernel ID_HUBS. Once ID_HUBS has terminated, all non-members, which are not hub nodes, are considered to be outlier nodes. This is illustrated in Fig. 7 and summarized in Algorithm 10.

Nodes in A.u in Fig. 7a which belong to a cluster as indicated by parent in Fig. 7b are replaced by a null value as seen in Fig. 7c. Nodes in A.v corresponding to non-null values in A.u are replaced by their parents. Pairs in A containing at least one null value (in light blue color) are removed, and A is sorted by A.u as seen in Fig. 7d. Then each pair in A is compared to the previous pair. When two pairs have the same reference node but different adjacent nodes (e.g. node 4), the parent of the reference node is changed to indicate that reference node is a hub (see Fig. 7e).

Algorithm 10. Non-member classification

```
host function classNonMembers(A.u, A.v, parent)
input: A.u, A.v, the adjacency lists, and parent, the array of component labels.
output: parent, updated to include hub and outlier labels.
using 2 * |E| parallel threads on the device execute:
begin
  device kernel ID OUTLIERS(parent, A.u)
    declare integer tid ← global thread ID
    if parent[A.u[tid]] ! = null then
      A.u[tid] ← null
    end if
end begin
|A.u| ← partition({A.u,A.v}, {A.u, A.v} + |E|, A.u ! = null)
using |A.u| parallel threads on the device execute:
begin
  device kernel ID ADJ CLUSTERS(parent, A.v)
    declare integer tid ← global thread ID
    A.v[tid] ← parent[A.v[tid]]
end begin
|A.u| ← partition({A.u,A.v}, {A.u, A.v} + |E|, A.v ! = null)
using |A.u| parallel threads on the device execute:
begin
  device kernel ID HUBS(A.u, A.v, parent)
  declare integer tid ← global thread ID
  declare integer bid ← tid mod block size
  declare integer u[], v[] in shared memory
  u[bid] ← A.u[tid]
  v[bid] ← A.v[tid]
  if u[bid] = u[bid – 1] and v[bid] ! = v[bid – 1] then
      parent[u[bid]] ← HUB
    end if
end begin
```

GPUSCAN 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. 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 the host memory. Now GPUSCAN has completed the entire process.

4 Results

Implementation of the GPUSCAN was written in C++ with CUDA and executed on two different machines which include the Geforce GTX 460 and the Geforce GTX 480. Both machines use the Intel quad-core i7 2.6 GHz processors

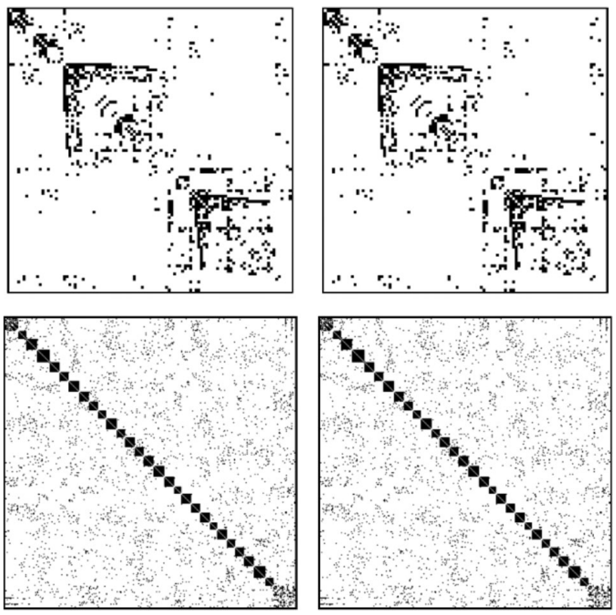

Fig. 8. Bitmap representations of the adjacency matrixes for two real world data sets. Up left is adjacency matrix of Political Books clustered by SCAN. Political Books clustered by GPUSCAN is up right. NCAA teams clustered by SCAN is bottom left. NCAA teams clustered by GPUSCAN is bottom right.

with 8GB of memory. We also re-implemented the optimized serial version of SCAN given in Algorithm 1 in C++.

4.1 Equivalence to SCAN

We tested results generated from GPUSCAN and SCAN for equivalence by primarily using two small real world networks and two large artificial networks: one SSCA 2 and one RMAT network with more than 524 thousand nodes (see Section 4.2 for details about these synthetic networks). One of the real world networks is the set of U.S. political books compiled by Valdis Krebs [22]. This network contains 105 books. As a network, nodes represent the books, and edges represent books frequently sold together through Amazon.com. The other real world network is based on the football game schedule of the National Collegiate Athletic Association (NCAA), with the 323 nodes representing NCAA teams and their opponents, and the edges representing games in the schedule. Results of clustering from SCAN and GPUSCAN are identical for all these networks, for the same set of parameters. Moreover, we tested equality on every network used in this study. Results were identical.

To easily see the equivalence instead of comparing one by one for hundreds of thousands of nodes, we represent the results in bitmap images and take the image difference. If results are the same, image difference should generate a completely white image. To do so, for each cluster found in the network, the members are sorted by node identifier, and given the minimum node identifier in the cluster as those nodes' cluster identifiers. 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 image as seen in Fig. 8. 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.

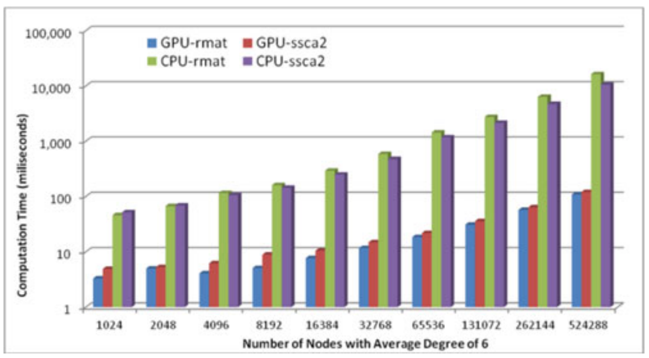

Fig. 9. Performance comparisons including computing similarity for synthetic networks with constant average degree 6 and varying number of nodes.

Clustering political books network with SCAN using $\varepsilon = 0.35$ 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 first row of Fig. 8, computing this clustering with GPUSCAN using the same parameters produces the exact same results. For NCAA network for instance, with the parameters ($\varepsilon = 0.5$, $\mu = 2$), SCAN produces 29 dense clusters, representing sub-conferences. Using the same parameters, GPUSCAN again produces the exact same clustering as SCAN does. This is shown in the second row of Fig. 8.

4.2 Performance Comparisons: GPUSCAN versus SCAN

In developing GPUSCAN, we sought an algorithm that was faster than SCAN with at most a linear time complexity. In this subsection, we measure computation times of SCAN and GPUSCAN for two cases, one where both structural similarity and clustering are computed, and the other where the cost of computing structural similarity is not included in comparisons. This is because, in the original SCAN paper, authors presumed that structural similarities are already computed.

GPUSCAN and SCAN are tested on both artificially created networks and real world networks. Two types of synthetic network generators were chosen for evaluating computation times: RMAT and SSCA#2. RMAT graphs are best suited to model real world graphs, having a large number of vertices with only a few outgoing edges, and a few vertices having many outgoing edges. SSCA#2 graphs consist of random sized cliques of vertices connected by a hierarchical distribution of edges among cliques. These network generators are included in the GTgraph suite and are available online [24].

A total of forty synthetic graphs were generated: ten RMAT graphs and ten SSCA#2 graphs have a constant 32,768 vertices of average degree varying from 10 to 100. Also, ten RMAT graphs and ten SSCA#2 graphs with a constant degree of 6 and with a number of vertices varying in powers of two from 1,024 to 524,288 are generated.

4.2.1 Performance Comparisons by Including Similarity Computations

Fig. 9 compares time to compute both structural similarity and clustering in network using SCAN and GPUSCAN on the synthetic networks with varying number of nodes. The

TABLE 1
Speedup Factors for Graphs Given in Figs. 9 and 10.

Average Degree 6			32,768 Nodes		
# of nodes	rmat	ssca2	degree	rmat	ssca2
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	**70**	214.2	224.7
131072	88.7	59.7	**80**	219.6	235
262144	111.1	74.1	**90**	235	250.5
524288	149.2	87.6	**100**	239.2	254.2

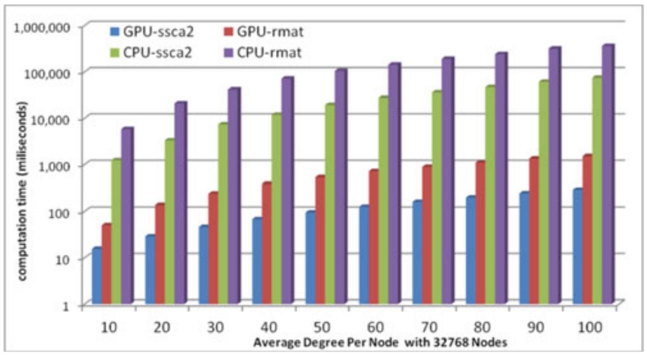

Fig. 10. Performance comparisons including computing similarity for synthetic networks with constant number of nodes, but varying average degree.

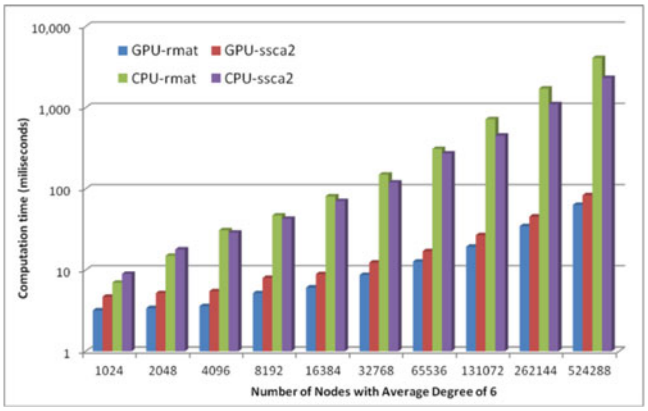

Fig. 11. Performance comparisons for synthetic networks with constant average degree of 6 and varying number of nodes.

networks used for comparison in Fig. 11 SSCA2 networks have varying average degree while the number of nodes remains constant. In these figures CPU corresponds to the sequential SCAN and GPU corresponds to GPUSCAN. Table 1 summarizes speedup factors according to results given in Figs. 9 and 10.

4.2.2 *Performance Comparisons with Given Pre-computed Structural Similarity*

Fig. 11 compares the time to compute clustering using SCAN and GPUSCAN on the synthetic networks with varying number of nodes. Fig. 12 illustrates the computation times of the two algorithms for the synthetic networks with constant number of nodes but varying average degree. In this case, the computation times in both figures do not include cost of computing structural similarity.

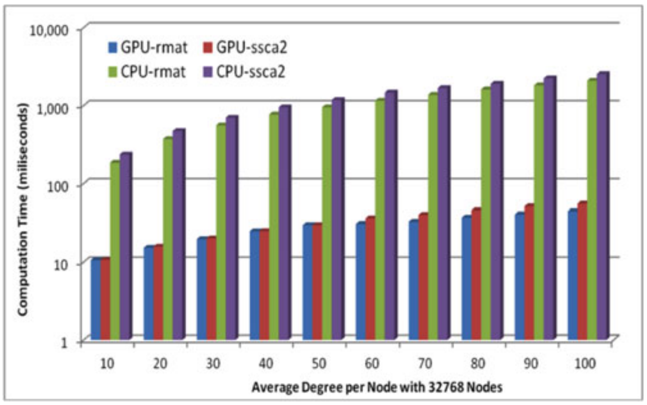

Fig. 12. Performance comparisons for synthetic networks with constant number of nodes, but varying average degree

TABLE 2
Speedup Factors for Graphs given in Figs 11–12

Average Degree 6			32,768 Nodes		
#of nodes	rmat	ssca2	degree	rmat	ssca2
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	5.4	**40**	30.9	38.2
16384	13.3	7.9	**50**	32	39.5
32768	17.3	9.7	**60**	37.5	40.4
65536	24.4	16	**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	**100**	46.6	45.7

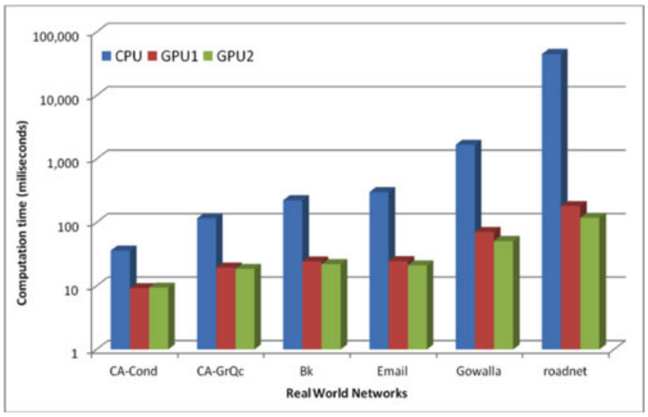

Fig. 13. GPUSCAN versus SCAN for real world networks, including calculating structural similarity.

Table 2 summarizes speedup factors for results given in Figs. 12 and 13. In Figs. 10, 11, 12, and 13 the computation times for both SCAN and GPUSCAN are shown to be approximately linear for calculating structural similarity and for clustering, regardless of whether the size of the network varies by the number nodes or by the average degree (affecting the total number of edges). The figures do show

TABLE 3
Timings for Calculating Structural Similarity and Clustering in Real World Networks

	CA-Cond	CA-GrQc	Bk	Email	Gowalla	Road
NODES	23K	5.2K	58K	37K	197K	2.0M
EDGES	186K	290K	214K	368K	950K	5.5M
CPU	118	581	3009	5192	61915	107333
GPU1	11.69	32	93.12	148.1	1593.2	293.4
GPU2	11.29	28.21	78.3	124.75	1213.37	209.75

that SCAN tends to be slightly superlinear, while GPUSCAN tends to be slightly sublinear in computation times. This disparity is likely due to the advantage of GPUSCAN using coalesced memory access as opposed to SCAN's random access model

4.2.3 Performance Comparisons on Real World Networks

We obtained six real world networks from Stanford's SNAP library [23] to evaluate the performance of SCAN and GPUSCAN. The GTX 460 and the GTX 480 were used to perform GPUSCAN whereas SCAN was performed on the CPU.

Fig. 13 illustrates and Table 3 shows the computation times for three test cases to compute both structural similarity and clustering. In Fig. 14 and Table 4, similarity metric is assumed to be pre-computed, so the computation times are for clustering only. The computation times given in Tables 3 and 4 include data transfer times from CPU to GPU or the other way around. Table 5 shows data transfer times for the 6 real world networks used in tests.

The first real world network we used is a collaboration network in Arxiv and covers scientific collaborations between authors' papers submitted to General Relativity and Quantum Cosmology (GrQc) category. This network consists of 5,242 nodes and 28,980 edges. The second network is again a collaboration network in Arxiv with 23,133 nodes and 186,936 edges. It covers scientific collaborations between authors of the papers submitted to Condense Matter (Cond) category. The third real world network is based on a social network called Brightkite (BK) where users share their locations. The network consists of 58,228 nodes and 214,078 edges. The fourth real world network is Enron email network which is formed by the email communications in Enron. There are 36,692 nodes and 367,662 edges within the network.

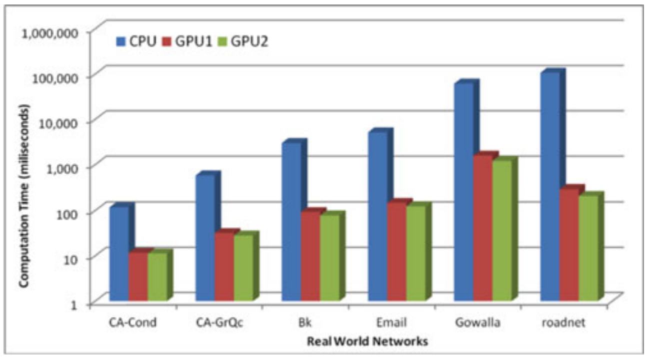

Fig. 14. GPUSCAN versus SCAN for real world networks, does not include calculating structural similarity.

TABLE 4
Timings for Calculating only Clustering in Real World Networks

	CA-Cond	CA-GrQc	Bk	Email	Gowalla	Road
NODES	23K	5.2K	58K	37K	197K	2.0M
EDGES	186K	290K	214K	368K	950K	5.5M
CPU	36	116	224	301	1680	45083
GPU1	9.21	19.46	24.29	24.61	70.78	181.4
GPU2	9.41	18.56	22	21.11	51.14	118.65

The fifth real world network we used in performance test is called Gowalla. Gowalla is also a location-based social networking website where users share their locations. This network consists of 196,591 nodes and 950,327 edges. The most impressive results were obtained on the sixth real world network, California road network (Road). In this network, intersections and endpoints of roads in California are represented by nodes and the roads connecting these intersections or road endpoints are represented by undirected edges. There are 1,965,206 nodes and 5,533,214 edges in the network.

Table 3 and 4 show detailed computation times in milliseconds for six real world networks. Table 5 shows host to device or device to host data transfer times in GPUSCAN. These additional data transfer times are parallel overheads that serial SCAN does not have. For instance, as seen from Table 3 GPUSCAN is approximately 513 times faster than SCAN for California road network dataset. Network size increases GPUSCAN's computational efficiency over SCAN increases too.

5 CONCLUSIONS

In this study, GPUSCAN—which is a parallel redesign of the SCAN network clustering algorithm for massively parallel CUDA-enabled graphics cards—is introduced. We have outlined the tasks of transforming the sequential algorithm designed for execution on the CPU into the GPU parallel algorithm. GPUSCAN simplifies SCAN's serial computations in to set of SIMD computations and also aligns memory accesses to maximally benefit from memory hierarchy provided in GPUs. The performance of GPU accelerated structural clustering has been shown to be much faster than the sequential CPU implementation. Calculating structural similarity and clustering 5.5 million edges of the California road network in just around 210 milliseconds—a rate that is approximately 513 fold faster than SCAN. The results also indicate that this speedup can be greater for networks with larger numbers of edges and/or nodes.

GPUSCAN's method for calculating structural similarity is much more computationally efficient than its sequential counterpart. In the parallel version, nearly all memory accesses are coalesced. Furthermore, GPUSCAN avoids idle threads, allowing the GTX 480 to perform 64 to 128 simultaneous comparisons for intersection counting and to

TABLE 5
Data Transfer Times in Milliseconds

	CA-Cond	CA-GrQc	Bk	Email	Gowalla	Road
To Device	1.62	0.367	3.62	3.08	15	43
To Host	1.78	0.352	3.81	3.1	15.1	48.5

complete structural similarity calculation for 512 edges at a time. GPUSCAN is also very efficient because it focuses on high memory bandwidth and avoids warp diversions.

GPUSCAN was developed with CUDA in mind; however, the CUDA-enabled GPU acts as a microcosm for a parallel distributed system. GPUSCAN is scalable and can be readily adapted to a concurrent and/or distributed environment, due to its focus on a coalesced and local resource access approach. In contrast, SCAN is based on the breadth first search which, although computationally efficient, tends to require a high frequency of random resource accesses, and quickly becomes I/O bound for very large networks. Our implementation is generic enough to be executed on any CUDA enabled GPU with compute capability of 2.0 or greater. GPUs with at least 1GB of device memory can structurally cluster networks several times larger than those documented in the performance analysis of this paper. However, the current implementation requires that for each step the necessary data structures exist entirely in the device memory. Hence, GPUSCAN currently is limited by the amount of device memory available on the GPU. However, it can be easily scaled up by distributing computations over multiple connected GPUs.

Acknowledgments

The authors would like to thank Dr. Mutlu Mete and Dr. Nurcan Yuruk for providing insight and expertise on SCAN algorithm.

References

[1] R. Guimera and L. A. N. Amaral, “Functional cartography of complex metabolic networks,” *Nature*, vol. 433, pp. 895–900, 2005.

[2] M. E. J. Newman and M. Girvan, “Finding and evaluating community structure in networks,” *Phys. Rev. E 69*, 2004.

[3] C. Ding, X. He, H. Zha, M. Gu, and H. Simon, “A min-max cut algorithm for graph partitioning and data clustering,” in *Proc. IEEE Int. Conf. Data Mining*, 2001, pp. 107–114.

[4] J. Shi and J. Malik, “Normalized cuts and image segmentation,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, pp. 888–905, Aug. 2000.

[5] A. Clauset, M. E. J. Newman, and C. Moore, “Finding community in very large networks,” *Phys. Rev. E 70*, 2004.

[6] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger, “SCAN: Structural clustering algorithm for networks,” in *Proc. SIGKDD*, 2007, pp. 824–833.

[7] 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. Knowl. Discov. Data Mining*, 1996, pp. 291–316.

[8] M. E. J. Newman, *Networks: An introduction*. Oxford, United Kingdom: Oxford University Press, 1st ed., ISBN 0199206651, 2010.

[9] D. J. Watts and S. H. Strogatz, “Collective dynamics of ‘small-world’ networks,” *Nature*, vol. 393, pp. 440–442, 1998.

[10] NVIDIA. (2013). CUDA programming guide. [Online]. Available: http://developer.download.nvidia.com/compute/cuda

[11] J. Nickolls and W. J. Dally, “The gpu computing era,” *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Apr. 2010.

[12] 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.*, vol. 30, no. 6, pp. 864–872, 2009.

[13] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the gpu using cuda,” in *Proc. HiPC, LNCS 4873*, pp. 197–208, 2007.

[14] NVIDIA. Cuda Thrust Library. (2014). [Online]. Available: http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf

[15] K. A. Hawick, A. Leist, and D. P. Playne, “Parallel graph component labelling with GPUs and CUDA,” in *Proc. Parallel Computing 36*, pp. 655–679, 2010.

[16] R. Niewiadomski, J. Amaral, and R. Holte, “A parallel external-memory frontier breadth-first traversal algorithm for clusters of work-stations,” in *Proc. IEEE Int. Conf. Parallel Processing*, pp. 531–538, 2006.

[17] J. Soman, K. Kothapalli, and P. J. Narayanan, “Some GPU algorithms for graph connected components and spanning tree,” *Parallel Process. Lett.*, vol. 20, no. 4, pp. 325–339, 2010.

[18] K. Wu, E. Otoo, and K. Suzuki, “Optimizing two-pass connected-component labeling algorithms,” *Pattern Anal. Appl.*, vol. 12, no. 2, pp. 117–135, 2009.

[19] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, “Computing connected components in parallel computers,” *Commun. ACM 22*, vol. 8, pp. 461–464, 1979.

[20] O. Kalentev, A. Rai, S. Kemmitz, and R. Schneider, “Connected component labeling on a 2D grid using CUDA,” *J. Parallel Distrubted Comput.*, vol. 71, pp. 615–620, 2011.

[21] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, “Fast minimum spanning tree for large graphs on the GPU,” in *Proc. HPG 09: Proc. Conf. High Perform. Graphics*, pp. 167–171, 2009.

[22] (2014). [Online]. Available: http://www.orgnet.com

[23] (2014). [Online]. Available: http://snap.stanford.edu

[24] GTgraph generator suite. (2014). [Online]. Available: http://www.cse.psu.edu/~madduri/software/GTgraph/

[25] J. Huang, H. Sun, J. Han, H. Deng, Y. Sun, and Y. Liu, “SHRINK: A structural clustering algorithm for detecting hierarchical communities in networks,” in *Proc. Conf. Inf. Knowl. Management*, pp. 219–228, 2010.

[26] L. Luo, M. Wong, and W. Hwu, “An effective GPU implementation of breadth-first search,” in *Proc. ACM 47th Design Autom. Conf.*, 2010, pp. 52–55.

[27] M. Hussein, A. Varshney, and L. Davis, “On implementing graph cuts on CUDA,” in *Proc. First Workshop General Purpose Process. Graph. Process. Units*, 2007, http://www.cs.umd.edu/projects/gvil/papers/hussein_GPGPU07

[28] Facebook. (2013, Nov.). User Statistics. [Online]. Available: http://newsroom.fb.com/content/default.aspx?NewsAreaId = 22

[29] D. Ediger, K. Jiang, E. J. Riedy, and D. A. Bader, “GraphCT: Multithreaded algorithms for massive graph analysis,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 11, pp. 2220–2229, Nov. 2013.

[30] D. Gregor and A. Lumsdaine, “Lifting sequential graph algorithms for distributed-memory parallel computation,” *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 423–437, 2005.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” in *Proc. Int. Conf. Manage. Data*, pp. 135–146, 2010.

[32] A. Buluç, E. Duriakova, A. Fox, J. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams, “High-productivity and high-performance analysis of filtered semantic graphs,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, pp. 237–248, 2013.

[33] A. Buluç and K. Madduri, “Graph partitioning for scalable distributed graph computations,” in *Proc. Graph Partitioning and Graph Clustering (Proc. 10th DIMACS Implementation Challenge), Contemporary Mathematics*, vol. 588, pp. 83–101, 2013.

[34] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, “A flexible open-source toolbox for scalable complex graph analysis,” in *Proc. SIAM Conf. Data Mining*, 2012, pp. 930–941.

Thomas Ryan Stovall received the BS degree in mathematics at University of Central Arkansas, Conway, AR, in 2012, where he is currently working toward the graduation degree in Department of Computer Science. His research interests are numerical computational models and parallel computing.

Sinan Kockara received the BS degree in computer engineering at Dokuz Eylul University, Izmir, Turkey, in 2001. He received the PhD degree in applied computing at University of Arkansas at Little Rock, AR, in 2008. Since 2008, he has been with the Department of Computer Science at University of Central Arkansas, Conway. He is currently an Associate Professor in the same department. His research interests include biomedical informatics, medical image processing and analysis especially in dermoscopy, surgical simulation development for virtual reality environments, parallel computing and GPGPU. He is a member of the IEEE since 2006. He is also a member of the Sigma Xi scientific research society.

Recep Avci received the BS degree in mathematics at Bogazici University, Istanbul, Turkey, in 2007. He received the MS degree in applied computing at University of Central Arkansas, Conway, AR, in 2012. He is currently working toward the PhD degree in the Bioinformatics program at University of Arkansas, Little Rock, and is working as a Research Assistant at University Arkansas for Medical Sciences. His research interest areas are natural language processing, topic modelling, text mining and semantic analysis in medical reports, image processing, parallel computations using CUDA.