International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 3, Number 2, pages 264–288, July 2013

Data Structures and Algorithms for Counting Problems on Graphs using GPU

Amlan Chatterjee
School of Computer Science, University of Oklahoma
Norman, OK, 73019, USA

Sridhar Radhakrishnan
School of Computer Science, University of Oklahoma
Norman, OK, 73019, USA

and

John K. Antonio
School of Computer Science, University of Oklahoma
Norman, OK, 73019, USA

Received: July 25, 2012
Revised: October 27, 2012
Revised: February 17, 2013
Accepted: March 13, 2013
Communicated by Akihiro Fujiwara

Abstract

The availability and utility of large numbers of Graphical Processing Units (GPUs) have enabled parallel computations using extensive multi-threading. Sequential access to global memory and contention at the size-limited shared memory have been main impediments to fully exploiting potential performance in architectures having a massive number of GPUs. After performing extensive study of data structures and complexity analysis of various data access methodologies, we propose novel memory storage and retrieval techniques that enable parallel graph computations to overcome the above issues. More specifically, given a graph $G = (V, E)$ and an integer $k <= |V|$, we provide both storage techniques and algorithms to count the number of: a) connected subgraphs of size k; b) k cliques; and c) k independent sets, all of which can be exponential in number. Our storage techniques are based on creating a breadth-first search tree and storing it along with non-tree edges in a novel way. Our experiments solve the above mentioned problems by using both naïve and advanced data structures on the CPU and GPU. Speedup is achieved by solving the problems on the GPU even using a brute-force approach as compared to the implementations on the CPU. Utilizing the knowledge of BFS-tree properties, the performance gain on the GPU increases and ultimately outperforms the CPU by a factor of at least 5 for graphs that completely fit in the shared memory and by a factor of 10 for larger graphs stored using the global memory. The counting problems mentioned above have many uses, including the analysis of social networks.

Keywords: Counting Subgraphs, GPU Computation, CUDA

1 Introduction

The continuous growth and availability of huge graphs for modeling Online Social Networks (OSNs), World Wide Web and biological systems has rekindled interests in their analysis. Social networking sites such as Facebook with 750 million users [15], Twitter with 200 million users ([16]) and LinkedIn with over 100 million users [5] are a huge source of data for research in the fields of anthropology, social psychology, economics and others. It is impractical to analyze very large graphs with a single CPU, even if multi-threading is employed.

Recent advancements in computer hardware have led to the usage of graphics processors for solving general purpose problems. Compute Unified Device Architecture (CUDA) from Nvidia [7] and StreamSDK from AMD/ATI are interfaces for modern GPUs that enable the use of graphics cards as powerful co-processors. These systems enable acceleration of various algorithms including those involving graphs [11].

The graph problems that we focus on in this paper involve counting the number of subgraphs that satisfy a given property. We are interested in counting the number of: a) connected subgraphs of size k; b) cliques of size k; and c) independent sets of size k. A naïve mechanism to perform this counting is to generate a combination of the nodes (node IDs), construct the induced subgraph with the same, and check if the desired property holds. Since the number of subgraphs that can be constructed by choosing k out of n nodes is ${}^nC_k$, this approach is combinatorially explosive. But it is noted that this naïve approach does lend itself to be executed in parallel by using a large number of threads.

The Nvidia Tesla C1060 system contains 30 streaming multiprocessors, each containing 8 GPU processors and 16 memory banks each of size 1K. These 240 processors have access to both memory banks in the streaming multiprocessor they reside in and the global memory. Using the global memory instead of the memory inside the streaming multiprocessor referred to as the shared memory, results in overall kernel performance deterioration [3]. Contention results when two or more threads access memory (same memory bank or global memory partition) at the same time. Such concurrent requests are simply queued up and processed sequentially.

Using only the shared memory, the total space available in the Nvidia system mentioned previously is 30×16 KB = 480 KB. In practice, the entire 480 KB might not be available due to storage of kernel parameters and other intrinsic values in it. However, this effect can be reduced by storing such data in the constant memory instead of using the shared memory.

Using a boolean adjacency matrix i.e., using single bits to represent data values of 0's and 1's, a graph of size up to 1982 can be stored in 480 KB. For undirected graphs, we need to store only the upper triangular matrix and hence can store a graph of size up to 2804. The main issue in storing a single graph in the shared memory, spread across all the streaming multiprocessors, is certain solution combinations may contain nodes whose adjacency information may be stored in the shared memory of different streaming multiprocessors. CUDA architecture (see Fig. 1) and the Nvidia implementation does not allow a processor in one streaming multiprocessor to access a memory bank belonging to another; as a result the graph cannot be stored in an arbitrary fashion across all streaming multiprocessors. Considering the shared memory in a single streaming multiprocessor of size 16 KB, we can store graphs of size 360 (or 512) using adjacency matrix (or upper triangular matrix) representation. In this case information relating to nodes in any combination remains in the shared memory of the same streaming multiprocessor and threads mapped to associated processors can access them for processing.

Rather than generating all possible combinations, we have devised techniques to reduce the number of combinations by considering nodes in k neighborhoods (nodes that are at most distance k from each other.) This is accomplished by considering a breadth-first search (BFS) tree, constructed in the host CPU, and nodes in any adjacent k levels only. Additionally, this BFS aided technique allows us to carefully split the graph in such a way that we can use the entire shared memory available across all the streaming multiprocessors and process larger graphs – even those that have to be stored externally. Using all streaming multiprocessors not only helps in processing larger graphs, but also allows the use of all the GPU processors thereby decreasing total execution time.

The outline of our paper is as follows. In Section 2, we present information on related work. In

Section 3, an overview of the GPU architecture is provided; memory hierarchy and access patterns are also discussed. Section 4 analyzes the proposed simple data structures used to fit adjacency information of graphs on the shared memory of all streaming multiprocessors. A naïve approach for counting connected subgraphs is discussed in Section 5. In Section 6 we present information on using BFS-tree and its properties to reduce the number of computations and storage requirements. Algorithms for splitting and solving problems on graphs that do not fit in the shared memory of a single streaming multiprocessor are discussed in Section 7. Section 8 discusses algorithms for other related problems such as finding the total number of k cliques and k independent sets. Implementation results for counting the number of connected subgraphs using various graph representations stored both in global and shared memory are presented in Section 9. Conclusion and future work is discussed in Section 10.

2 Related work

Using efficient data structures to store graphs for computation on both CPUs and GPUs have been studied extensively. Various modifications of the adjacency matrix and adjacency list data structures are considered.

Katz and Kider [13] and Buluc et al. [4] proposed storing graphs on the GPU by dividing the adjacency matrix into smaller blocks. The required blocks are loaded in the memory, and after computation, are replaced by the next set of blocks. This representation still uses the adjacency matrix, and might include data which is not required for computations based on locality information.

Frishman and Tal [8] propose representing multi-level graphs using 2D arrays of textures. They propose partitioning the graph in a balanced way, by identifying geometrically close nodes and putting them in the same partition. Since the partitions are based on locality information and balanced, it makes use of the GPUs data parallel architecture. But partitioning the graph itself is a hard problem.

For sparse matrices, the Compressed Sparse Row (CSR) representation is useful [9]. Also, representing the edge information using arrays to store the out-vertex and in-vertex numbers saves space for sparse matrices [12]. This method is better suited for directed graphs.

Bader and Madduri [1] proposed a technique where different representations are used depending on the degree of the vertices. This is relevant for storing graphs that exhibit the small-world network property, where vertices have an unbalanced degree distribution, with majority of vertices having small degrees and a few vertices are of very high degree.

Harish and Narayanan [11] describe the use of a compact adjacency list, where instead of using several lists, the data is stored in a single list. Using pointers for each of the vertices' adjacency information, data for the entire graph is kept in a single one dimensional array, which can be significantly large to be stored in the shared memory, and has been implemented in the global memory in their paper.

Bordino et al. [2] have developed a technique for counting subgraphs of size 3 and 4 for large streaming graphs. The counting algorithm is sequential in nature and can be used in many applications.

In this paper, in addition to using the breadth-first search (BFS) information to carefully split the graph for processing, we introduce data structures for storing nodes and their adjacency information that are in contiguous levels of the BFS-tree. We propose both simple and modified data structures using least number of bits in addition to exploiting the symmetric property of undirected graphs. The modified data structure is similar to the one proposed by Harish and Narayanan [11], but with improvements, including the use of fewer bits in the general case and also using more than one array to store the entire adjacency data for the graph, thereby adhering to stricter memory requirement constraints.

This paper is an extension of our previous work on techniques to solve counting problems in graphs [6]. In addition to the results presented in [6], detailed analysis of various other data structures for storing graph data is discussed in this paper. Also, time complexity comparison for different operations involving data access from the various data structures are also included. Results show-

ing time comparison for implementation of algorithms using both CPU and GPU have also been provided.

3 GPU architecture and CUDA programming model

An overview of the GPU architecture and CUDA programming model is presented here. In addition, the different levels of the memory hierarchy available on the GPU and access patterns for the same are also discussed.

3.1 GPU architecture

The GPU, also referred to as the device, consists of streaming multiprocessors, and multiple levels of memory hierarchy. The GPU used for the experiments in this paper is the Nvidia Tesla C1060. It has 30 Streaming Multiprocessors (SMs), with 8 cores in each of them giving a total of 240 cores. Other GPU cards from Nvidia, like C2050 and C2070 contains a total of 448 cores in them. Computation heavy tasks are transferred from the CPU to the GPU, and by using all the available processor cores and multithreading options, problems with large number of computations can be solved efficiently. The GPU architecture of Nvidia Tesla C1060 is shown in Fig. 1. Data required for computations is transferred to the GPU from the CPU using the global or device memory on the GPU; similarly computed results are transferred back from the GPU to the CPU using the same.

3.2 Programming model

CUDA provides a general-purpose parallel programming model. The programming language for CUDA extends C and defines C-like functions, called *kernels*, which are executed in parallel by different threads on the GPU. Threads are grouped together in blocks, which can be either one, two or three dimensional, and each thread within a block is identified by its unique *threadID*. All the blocks of threads, identified by *blockID*, form a one or two dimensional grid. Threads within a block can co-operate among themselves through the shared memory and synchronize their execution to coordinate memory accesses. Blocks of threads are assigned to the streaming multiprocessors by the CUDA scheduler. Data is transferred from the CPU to the GPU and back using memory copy functions.

3.3 Memory hierarchy and access patterns

The memory hierarchy of the GPU consists of global memory (also referred to as device memory), shared memory, constant memory, texture memory and registers. For Nvidia C1060, the size of the device memory is 4 GB, constant and texture memory are each 64 KB, and shared memory is 16 KB. The shared memory is further divided into 16 banks as shown in Fig. 1. The global memory is the largest and also has the highest access latency. The on-chip shared memory has significantly faster access compared to the global memory. But, when data is accessed from the same bank in the shared memory, either the same element or different elements, then there is a bank conflict leading to increased latency thereby leading to a performance loss. The only exception being the case where all the threads access the same element in the shared memory leading to a broadcast. The constant memory is also faster than the global memory, but it is a read-only memory and data stored in it cannot be modified. Fig. 2 shows the different levels of the memory hierarchy available on a GPU.

3.4 Using shared memory for better performance

As evident from the CUDA programming guide [7] and related work [3], and also confirmed from our experiments, the shared memory is a lot faster than the global memory. So, for better performance the data is stored, accessed and modified from the shared memory whenever possible. As the shared memory in each streaming multiprocessor has a capacity of 16 KB, and different streaming multiprocessors can have different data stored in each of them, the total size of data being processed

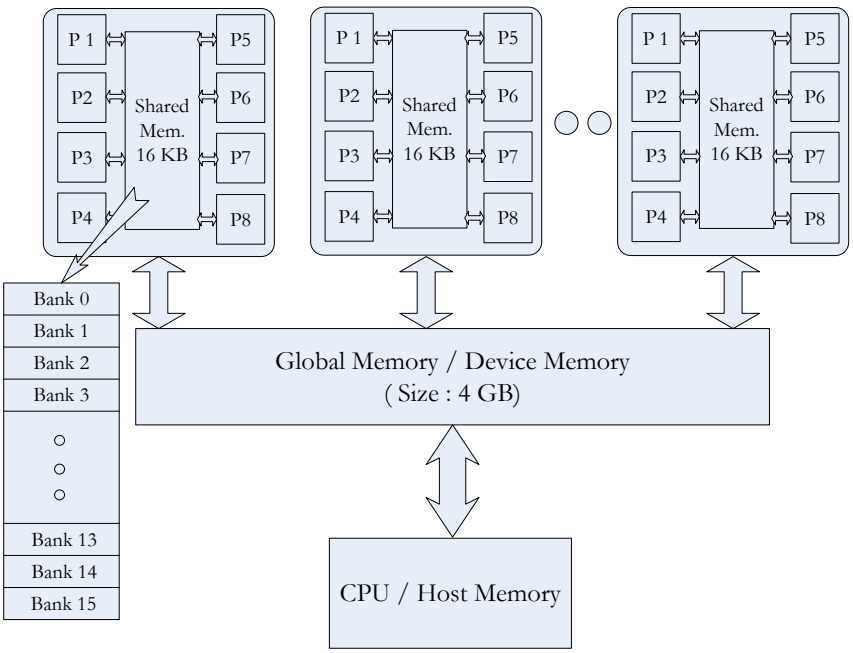


Figure 1: GPU architecture (C1060) [7]

is limited by that available on the 30 streaming multiprocessors. Since registers are also located on-chip, the memory access latency is much lower compared to that of the global memory. Automatic variables declared in a cuda kernel are placed into registers. However, since we are using large arrays and require certain computations on the data, which can be done efficiently using indices based on threads, storing the data in the registers was not considered in this paper.

4 Simple data structures for storing the graph information

Various data structures can be chosen to store the adjacency information of graphs in the GPU memory. Each of the data structures have different storage requirements. There are several operations on graphs that need to access the stored data using one of the available data structures. A memory efficient data structure might have worse time complexity when it comes to accessing the required data for computations. Hence, there exists a trade-off between memory requirement and access time complexity, when choosing the appropriate data structure for storing the adjacency information of the graphs. In this Section, we analyze the space required by different data structures and also the time complexity for performing common operations on graphs using the same. It must be noted that throughout the paper, for calculations we use boolean data and it consists of a single bit, and does not refer to the data type available in programming languages.

4.1 Adjacency Matrix

For a graph $G = (V, E)$ with $|V| = n$, the size of adjacency matrix is n^2 bits, where each edge is stored using a single bit. To fit the adjacency matrix in the shared memory, the space required must be less than or equal to that of the desired level in the memory hierarchy of the GPU. Considering the architecture available on the Nvidia 10-series GPUs, for example C1060, the size of the shared memory is 16 KB. Hence, to satisfy the above constraint, $n^2 \leq 131,072$ (16 KB $= 16 \times 1024 \times 8$ bits $= 131,072$ bits), which gives $n \approx 360$. Therefore, using the adjacency matrix representation, the size of the largest graph that can be kept in the shared memory is 360 (assuming all shared memories in the different streaming multiprocessors contain identical data.)

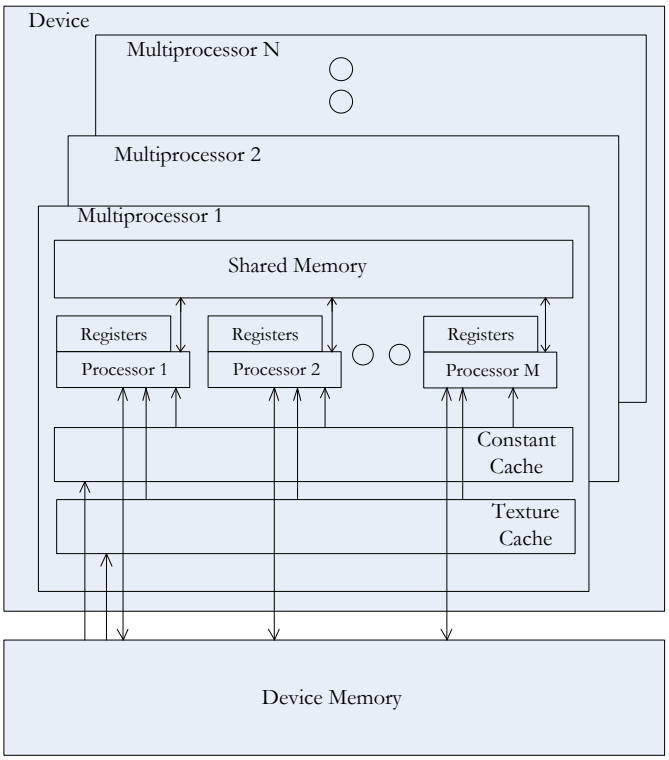


Figure 2: GPU memory hierarchy [7]

4.2 Upper Triangular Matrix

The adjacency matrix representation contains redundant information, and those can be eliminated to reduce the storage requirements. For undirected graphs, values (i, j) and (j, i) are identical. So, storing only the Upper Triangular Matrix (UTM) of the adjacency matrix is enough, which requires $\frac{n\times(n+1)}{2}$ bits. So, the largest graph that can be kept in the shared memory using the UTM representation is 511. As all the values of $(i, i) = 0$, using the Strictly UTM representation (S-UTM) (i.e., without the data on the diagonal), size of the largest graph that can be kept in the shared memory is 512. Although the shared memory spans across 16 banks, it is preferable to store data for any specific node within a single bank thereby avoiding potential memory contention and reducing overall execution time. With the above requirement, the number of nodes that can fit in the shared memory is reduced to 506, where data is kept in increasing order of the node numbers. Also, using UTM representation, the number of nodes' data in each of the bank (size 8192 bits) varies, as shown in Table 1.

In the previous approach, due to unbalanced distribution of nodes, threads assigned to operate on banks with more nodes would have to do significantly more work than the threads accessing banks with less number of nodes. On the other hand, if threads access a constant number of nodes it will result in inefficient memory utilization, thus limiting the overall size of graph that can be stored.

For load balancing the distribution can be done as follows. Using S-UTM representation different rows have different amount of data (see Table 2). To make the structure rectangular (see Table 3), the space gained by not storing redundant information in any row in the upper part of the S-UTM can be filled up by a corresponding row from the lower part. When n is even, the space gained in row i is filled with data values from row $n - i$ (see Table 3); when n is odd, the corresponding space in row i is filled with data from row $n - (i + 1)$ (see Table 5). In general, for any value of n the number of rows of data is reduced from n to $\lfloor \frac{n}{2} \rfloor$. This is called *Balanced* S-UTM (B-S-UTM),

Table 1: Distribution of nodes in the banks

Bank #	**Nodes in the Bank**	**# of Nodes**	**Space Required in bits**
0	0-15	16	7976
1	16-31	16	7720
2	32-48	17	7922
3	49-66	18	8073
4	67-85	19	8170
5	86-104	19	7809
6	105-124	20	7830
7	125-146	22	8151
8	147-169	23	8004
9	170-194	25	8100
10	195-221	27	8046
11	222-251	30	8085
12	252-285	34	8075
13	286-325	40	8020
14	326-378	53	8162
15	379-505	127	8128

where all rows have the same amount of data, each corresponding to that of 2 nodes. The above method is similar to "rectangular full packed" in dense linear algebra [10]. With the desire to store an entire row of data in a single streaming multiprocessor, this scheme also ensures all banks have equal number of nodes in them thereby achieving load-balancing. Using this scheme, the maximum number of rows that can be kept in a single bank is $\frac{(1024\times 8)}{511} \approx 16$. As there are 2 nodes' data in each row, the total number of nodes' data in each of the banks is 32. Therefore, the total number of nodes that can be kept in this manner in the 16 banks is $32 \times 16 = 512$.

Table 2: S-UTM for even number of nodes

–	a	b	c	d	e	f	g
–	–	h	i	j	k	l	m
–	–	–	n	o	p	q	r
–	–	–	–	s	t	u	v
–	–	–	–	–	w	x	y
–	–	–	–	–	–	z	ϕ
–	–	–	–	–	–	–	ψ
–	–	–	–	–	–	–	–

Table 3: S-UTM with load balanced approach for even number of nodes

a	b	c	d	e	f	g
h	i	j	k	l	m	ψ
n	o	p	q	r	z	ϕ
s	t	u	v	w	x	y

The number of simultaneous thread executions is limited by the number of GPU processors in a streaming multiprocessor. Each thread is allocated a set of combinations of nodes (with cardinality k) and a thread is to determine if the desired property (e.g., do they form a connected subgraph?) holds. In order to use all available GPU processors in other streaming multiprocessors, we have to duplicate the graph and place it on all the shared memories on other streaming multiprocessors.

Table 4: S-UTM for odd number of nodes

–	a	b	c	d	e	f
–	–	g	h	i	j	k
–	–	–	l	m	n	o
–	–	–	–	p	q	r
–	–	–	–	–	s	t
–	–	–	–	–	–	u
–	–	–	–	–	–	–

Table 5: S-UTM with load balanced approach for odd number of nodes

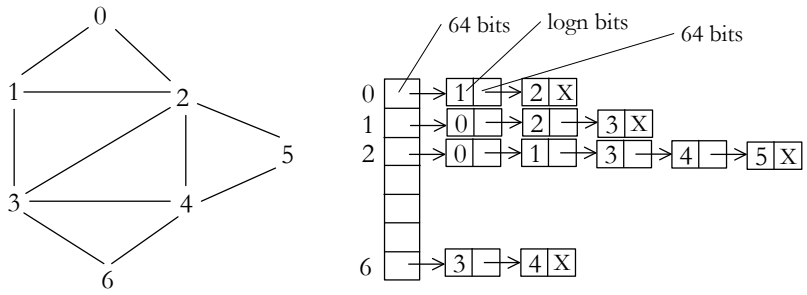
a	b	c	d	e	f	u
g	h	i	j	k	s	t
l	m	n	o	p	q	r

Care must be taken to ensure each thread is given a unique set of combinations to test, to avoid duplication in work.

The sets of combination of k nodes are allocated to each streaming multiprocessor as follows. Since the shared memory in each streaming multiprocessor can store up to 512 nodes, it can be assumed that there are 512 sets of combinations, each starting with a unique node number. We allow the first 29 streaming multiprocessors to operate on 17 unique sets of combinations each and the last streaming multiprocessor operates on the remaining 19 sets ($17 \times 29 + 19 = 512$). In each streaming multiprocessor, depending on the number of threads the unique sets are uniformly divided to be processed.

4.3 Adjacency List Using Array of Linked Lists

Other than the adjacency matrix, adjacency list is also a common data structure used to store graph information. There can be various modifications in the implementation of the adjacency list, and each has different memory requirements and data access complexity. In this sub-section, we study the implementation of the adjacency list using an array of linked lists, also referred to in here as AL-AL, and is shown in Fig. 3.



An example graph (left) with the corresponding adjacency list representation (right) using Array of Linked Lists (AL-AL)

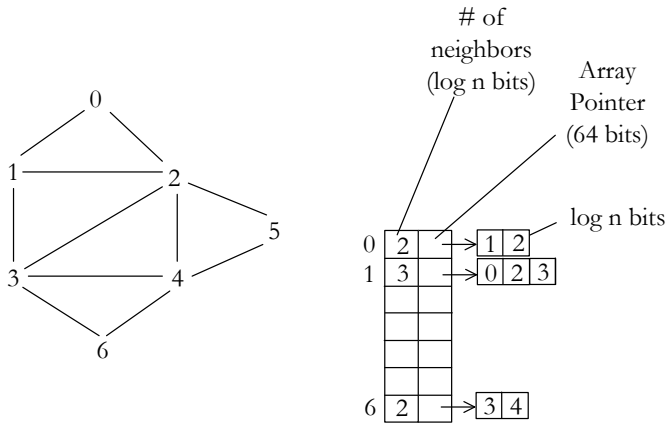
Figure 3: Adjacency List Using Array of Linked Lists

Let the total number of nodes in the graph be n and total number of edges be m. Here, all the nodes are stored in an array; the identifiers of the nodes given by the indices of the array

location. Each array element contains a pointer to the starting node of the linked list representing the neighbors of the corresponding node i.e., the edge information. In the example graph shown in Fig. 3, node 0 is connected to nodes 1 and 2. Therefore, a pointer to node 1 is stored in the array index corresponding to node 0. The storage for node 1 contains the identifier for the node, and a pointer to the next neighbor i.e., node 2. Since, there are no more neighbors for node 0, the pointer associated with node 2 contains an invalid marker, denoted in Fig. 3 by "X", which usually has an identifier value of -1. Now, each of the edges would be stored two times, once for each end vertex. Considering a 64-bit machine, each of the pointers require 64 bits. So, the space needed to store the array containing the nodes is given by $n \times 64$ bits. Now, each of the edges are represented by the end vertex number, and also contains a pointer to the next edge information if there exists another edge for the node under consideration. Since there are n nodes in the graph, representing a node number requires $\log n$ bits. Hence, for storing each of the edges, the space required is $(\log n + 64)$ bits, giving a total of $2m \times (\log n + 64)$ bits for all the edges. Therefore, the total size required for this representation is given by $n \times 64 + 2m \times (\log n + 64)$ bits.

4.4 Adjacency List Using Array of Arrays

In the adjacency list representation using array of linked lists, there are too many pointers involved in the implementation, thereby increasing the size of the data structure. Using arrays to group data together replacing pointers can reduce the memory required, and this is referred to as adjacency list using array of arrays (AL-AA), and is shown in Fig. 4.



An example graph (left) with the corresponding adjacency list representation (right) using Array of Arrays (AL-AA)

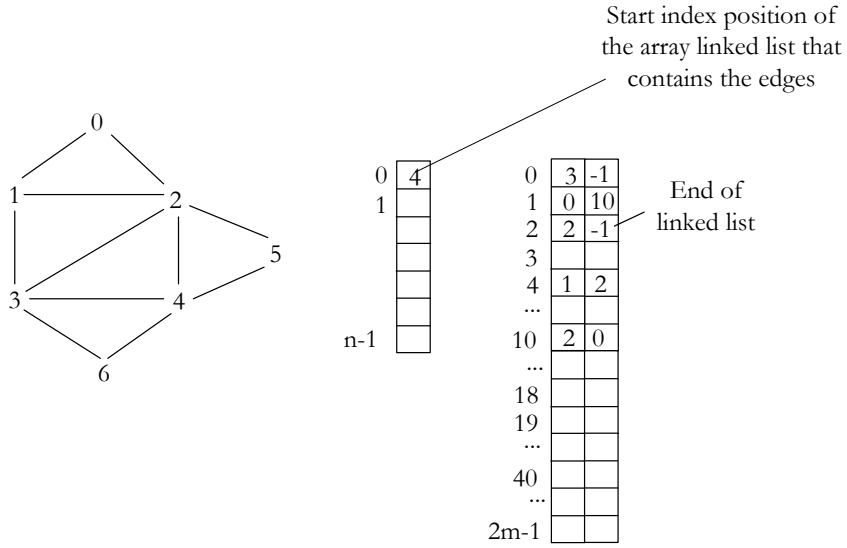
Figure 4: Adjacency List Using Array of Arrays

Here the edges from each node are stored using arrays instead of linked lists. Since there are no pointers involved between edges, to determine the size of the array, the number of neighbors for each of the nodes need to be stored. In this case, all the nodes in the graph are stored in an array, along with the number of neighbors and in addition there is a pointer for each of the nodes to the array containing its neighbors. In the example graph shown in Fig. 4, node 1 is connected to 3 nodes, numbered 0, 2 and 3. Therefore, for node 1, a value of 3 is stored corresponding to the number of neighbors, and also a pointer to the array of its neighbors, which contains the identifiers for the nodes 0, 2 and 3. Storing the array with the number of neighbors and pointers for the n nodes requires $n \times (\log n + 64)$ bits. Additionally, the total space required for all the arrays containing information about the m edges is $2m \times \log n$ bits. Therefore, the total size required for this representation in bits is given by $n \times (\log n + 64) + 2m \times \log n$.

An improved version of this data structure that requires less memory is one which does not store the number of neighbors for each of the nodes. However, this information is required to retrieve the adjacency data, and must be available during computation. This can be achieved by using the sizeof() function on the array containing the neighbors to find the required number at runtime. Hence, the array of arrays variant with the sizeof() function requires $n \times 64 + 2m \times \log n$ bits.

4.5 Adjacency List Using Array Implementation of Linked Lists

In the adjacency list representation using array of arrays, there are still pointers involved, and that requires significant amount of space. Instead of using pointers, the information for the edges can be stored using an array and relevant identifiers. So, the linked list pointers are replaced by using arrays. This representation is referred to as the adjacency list using array implementation of linked lists (AL-ALL), and is shown in Fig. 5.



An example graph (left) with the corresponding adjacency list representation (right) using Array Implementation of Linked List (AL-ALL)

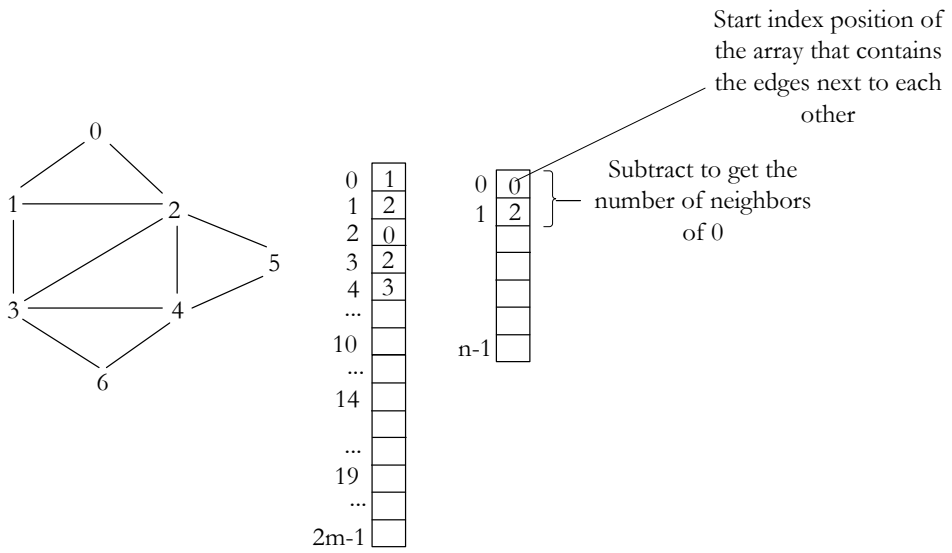
Figure 5: Adjacency List Using Array Implementation of Linked Lists

Here, both the nodes and edges are stored in arrays. Each element for the node array contains the starting index position in the array containing the edges. Since there are $2m$ indices in the edge array, the values stored in the node array each require $\log 2m$ bits, giving a size of $n \times \log 2m$ bits for the node array. The edges are represented by a pair of elements in the edge array. The first element contains the end vertex number of the edge, and the second element contains the index position of the next edge information in the array, or an identifier value of "-1" which indicates the end of the linked list. As there are n nodes, the first element requires $\log n$ bits, and for the $2m$ indices possible, the second element requires $\log 2m$ bits. So, the size of the edge array is $2m \times (\log n + \log 2m)$ bits. Therefore, the total size required for this representation in bits is given by $n \times \log 2m + 2m \times (\log n + \log 2m)$.

Now, the data representing the edges can be stored in any order in the array implementing the linked list. In the example graph shown in Fig. 5, the neighbors of node 0 are stored starting at position 4 in the array just to illustrate the fact that data can be stored at any location in the array as the pointers would correctly determine the location of the next available data. Node 1, which is a neighbor of node 0 is stored at location 4, and contains a pointer to location 2. Location 2

contains the identifier of the other neighbor of node 0 i.e., node 2 and contains an identifier "-1" for the pointer indicating the end of the list since there are no more neighbors for node 0.

4.6 Adjacency List with Edges Grouped



An example graph (left) with the corresponding adjacency list representation (right) with edges grouped (AL-EG)

Figure 6: Adjacency List with Edges Grouped

In the array implementation of linked lists, the edges are stored in an array using arbitrary indices which requires storing the next index position for each of the edges. This can be improved by storing all the edges next to each other. In this manner, neither the total number of edges for each of the node nor the next index positions are required. As before, both the nodes and edges are stored in arrays. The node array elements point to the starting index position of the edges in the edge array, and all the edges are stored in consecutive locations. The total number of edges for each of the nodes can be easily calculated by subtracting the starting index position of the current node with that of the next node. This representation, similar to the one described in [11] as compact adjacency list, is referred to as the adjacency list with edges grouped (AL-EG), and is shown in Fig. 6.

In the example graph shown in Fig. 6, node 0 has two neighbors, nodes 1 and 2. This information is stored in the edge array using consecutive locations starting from 0. Therefore, the node array element corresponding to node 0 contains the value of location 0. For node 1, the neighbors 0, 2 and 3 are stored in the edge array starting at location 2, and this value is stored in the node array. Using the information in the node array for nodes 0 and 1, the number of neighbors for node 0 can be found by subtracting the corresponding value of node 0 from that of node 1.

As in the case of array implementation of linked lists, the space required for the node array is $n \times \log 2m$ bits. For the edge array, there are $2m$ elements, and each require $\log n$ bits. So, the size of the edge array is $2m \times \log n$ bits. Therefore, the total size required for this representation in bits is given by $n \times \log 2m + 2m \times \log n$.

4.7 Comparison of memory requirements for the different data structures

In the above sub-sections, different data structures that can be used to represent the adjacency information for graphs are discussed. Gradual modifications are made to reduce the space requirements by eliminating the need for pointers and rearranging the data. For a graph with n nodes and m edges, space requirements for the different data structures can be summarized as given in Table 6.

Table 6: Comparison of memory requirements for the different data structures

Data Structure	**Total bits**
Adjacency matrix	n^2
S-UTM	$n \times (n-1)/2$
Array of linked lists	$n \times 64 + 2m \times (\log n + 64)$
Array of arrays	$n \times (\log n + 64) + 2m \times \log n$
Array of arrays with sizeof()	$n \times 64 + 2m \times \log n$
Array implementation of linked lists	$n \times \log 2m + 2m \times (\log n + \log 2m)$
Adjacency list with edges grouped	$n \times \log 2m + 2m \times \log n$

4.8 Operations on graphs

For solving various problems on graphs, different operations need to be performed on the graph data. In this sub-section we consider two types of operations on graphs - a) Query operations, and b) Update operations. The different types of query operations can be the following:

- $N(v)$: List the neighbors of node v.
- $D(v)$: Find the number of neighbors of node v.
- $Edge(u, v)$: Determine if there is an edge between vertices u and v.

The different types of update operations can be listed as follows:

- $Add(v)$: Add a vertex v.
- $Add(u, v)$: Add an edge between the vertices u and v.
- $Remove(v)$: Remove the vertex v and all corresponding edges (u, v).
- $Remove(u, v)$: Remove the edge (u, v).

As mentioned earlier, there is a trade-off between the space required and access time complexity for different data structures. The various query and update operations provide a measure for the time complexity. For a given graph G with degree $\delta(G)$ consisting of n nodes and m edges, Table 7 provides a summary of the comparison of the time complexity for the operations on graphs using the data structures discussed in this Section.

5 Counting connected subgraphs: Naïve approach

Given a graph $G = (V, E)$ with $|V| = n$ and a value k as input, the problem is to count the number of connected subgraphs of size k in G. This can be done using a brute-force approach. The procedure is as follows. The nodes of the graph are numbered from 0 to $n-1$. All possible combinations of size k are generated one by one from n nodes. For each of the generated combinations, a breadth-first search (BFS) is performed, and if all the nodes are reachable starting from the first node, then the subgraph is connected, otherwise it is not. For a complete graph, the total number of such connected subgraphs is ${}^nC_k$, and this is the number of combinations that is required to be tested for any graph.

Table 7: Comparison of time complexity for operations on graphs using different data structures

Data Structure	***N(v)***	***D(v)***	***Edge (u, v)***	***Add(v)***	***Add (u, v)***	***Remove (v)***	***Remove (u, v)***
Adjacency matrix	n	n	1	n^2	1	n^2	1
S-UTM	n	n	1	n^2	1	n^2	1
Array of linked lists	$\delta(G)$	$\delta(G)$	$\delta(G)$	n	1	$n+m$	$\delta(G)$
Array of arrays	$\delta(G)$	1	$\log \delta(G)$	n	$\delta(G)$	$n+m$	$\delta(G)$
Array of arrays with sizeof()	$\delta(G)$	1	$\log \delta(G)$	n	$\delta(G)$	$n+m$	$\delta(G)$
Array implementation of linked lists	$\delta(G)$	$\delta(G)$	$\delta(G)$	n	1	$n+m$	$\delta(G)$
Adjacency list with edges grouped	$\delta(G)$	1	$\log \delta(G)$	n	m	$n+m$	m

The node numbers of the subgraphs are not stored for space constraints; only the total count for the connected subgraphs is reported. However, in this approach the total number of combinations to be checked is huge, and this can be effectively reduced as discussed in the following Section.

6 Using BFS-tree information to reduce the number of computations: Advanced approach

Considering a breadth-first search (BFS) representation of the graph, nodes chosen in any combination must be in k adjacent levels, otherwise the subgraph containing the nodes in the combination will not be connected. For example, let $k = 3$ and a combination contain the nodes 10, 12, and 14 at levels 4, 5, and 7, respectively in the BFS-tree. It is possible for nodes 10 and 12 to be connected by an edge since they are in adjacent levels. For the sake of discussion, assume there is an edge (10, 12) in the graph. It follows that there cannot be a edge (10, 14), for otherwise the level of node 14 in the BFS-tree must be 3, 4, or 5. A similar argument can be made for the edge (12, 14) and hence the graph induced by those vertices is not connected. From the above reasoning it is evident that using the graph's BFS-tree can be an effective tool in reducing the number of combinations to be tested. We will further examine this in the following subsections.

6.1 BFS-tree node numbering

The nodes in the BFS-tree are numbered in the order they are visited following a breadth-first search of the graph. Any arbitrary node is chosen as the starting or root node, and is numbered 0 and belongs to the first level. All nodes that are neighbors of the first node belong to the second level. Similarly, any unvisited node that is neighbor of the nodes in the *previous* level, belong to the *next* level.

6.2 BFS-tree properties and applications

The following are some of the properties of BFS-tree that are useful in the study of graphs:

1. If two nodes are neighbors in the original graph, their level numbers cannot differ by more than one. Let v_i and v_j be adjacent nodes belonging to levels l_{v_i} and l_{v_j} respectively in the BFS-tree. So, from this property we have $|l_{v_i} - l_{v_j}| \leq 1$.

2. Any node in a level with a parent numbered α can also be neighbors with nodes numbered greater than α in the level of α, but not less. Let node v_i be the parent of node v_j and Δ_j be the set of neighbors of v_j. Therefore, for any node v_t where $l_{v_i} = l_{v_t}$, $v_t \notin \Delta_j \forall t < i$.

3. The structure and *height* of the BFS-tree depends on the choice of the starting or root node.

6.3 Reducing number of combinations to be tested

In the case of purely random distribution of nodes all possible combinations must be tested. So, for n nodes and subgraphs of size k, the total number of combinations to be tested is ${}^nC_k$. If $n = 360$ and $k = 10$, for example, the corresponding value is ${}^{360}C_{10} \approx 8.88 \times 10^{18}$.

In the case where a BFS-tree of the graph fits in the shared memory, using its properties the number of combinations to be tested can be drastically reduced. The idea here is to test for combinations with nodes in each of the consecutive k levels of the graph. Let $n = 360$ and $k = 10$, and the number of nodes in each level of the graph be 20. Then the total number of levels L in the graph is $\frac{360}{20} = 18$. Therefore, the number of consecutive k levels is $L - k + 1 = 9$. Now, for each of these set of levels, k nodes are to be chosen. The number of combinations to be tested taking different number of levels at a time is given as follows:

Considering 1-level i.e., for each of the different levels: ${}^{20}C_{10} = 184,756 \approx 1.84 \times 10^5$. Considering 2-levels: $({}^{20}C_1 \times {}^{20}C_9 + {}^{20}C_2 \times {}^{20}C_8 + {}^{20}C_3 \times {}^{20}C_7 + {}^{20}C_4 \times {}^{20}C_6) \times 2 + {}^{20}C_5 \times {}^{20}C_5 = 847,291,016 \approx 8.47 \times 10^8$. Considering for 3-levels: $72,851,600,250 \approx 7.28 \times 10^{10}$. Similarly, we can calculate the number of combinations taking 4 levels to 10 levels at a time.

Considering the general case, where there is a total of n nodes divided among L levels, such that each level consists of $\frac{n}{L}$ nodes say p. Then, the number of combinations to be checked for k node subgraph is: $\sum {}^pC_{n_1} \times^p C_{n_2} \times \cdots \times^p C_{n_k}$ such that $\forall n_i \geq 1, \exists n_1 + n_2 + \cdots + n_k = k$.

When taking 1-level at a time, the number of combinations to be tested is $\approx 1.84 \times 10^5$. But, all available L levels will be tested for these many combinations when they are considered individually. So, total number of combinations when considering all the 1-levels is $\approx 1.84 \times 10^5 \times L$. Similarly, taking 2-levels at a time, the number of combinations to be tested is $\approx 7.28 \times 10^{10}$. But, there are $L - 1$ such combinations of consecutive 2-levels. So, the total combinations when considering the contribution of all the 2-levels is $\approx 8.47 \times 10^8 \times (L - 1)$. There are correspondingly $L-2$ combinations for consecutive 3-levels, $L - 3$ combinations for 4-levels and so on. In general, there are $L - (k - 1)$ such combinations for consecutive k levels. Therefore, the total number of combinations to be tested is given by: $1.84 \times 10^5 \times 18 + 8.47 \times 10^8 \times 17 + 7.28 \times 10^{10} \times 16 + 1.35 \times 10^{12} \times 15 + 9.82 \times 10^{12} \times 14 + 3.5 \times 10^{13} \times 13 + 6.9 \times 10^{13} \times 12 + 7.6 \times 10^{13} \times 11 + 4.3 \times 10^{13} \times 10 + 1.02 \times 10^{13} \times 9 = 2.8 \times 10^{15}$ i.e., about $2,800$ trillion. The number of combination testing decreases by $\approx 8.87 \times 10^{18}$.

6.4 Storing the graph with BFS Information

In order to process the reduced combinations described in the previous subsection, streaming multiprocessors should be responsible for generating only relevant combinations, by knowing the number of nodes in each level and BFS numbering of the nodes. The entire graph can be stored using any of the efficient data structures discussed in the previous section. For the purpose of discussion, let the graph be stored using S-UTM representation along with the information of the number of nodes at each level. Instead, it may be beneficial in some cases to store adjacent levels of the BFS-tree. Thus, except for the node in the root level for each set of adjacent levels, there will be a S-UTM data structure along with the starting node number and the total number of nodes. This representation is called S-UTM-ADJ. Similarly, such corresponding representations exists for the other data structures too.

In addition to the S-UTM-ADJ we have devised another data structure called *Parent Array Representation* (PAR) wherein each node keeps information of its parent along with adjacent nodes

in both the parent's and the same level as a list. By keeping additional information we can further reduce the space requirements as shown below.

6.4.1 Parent Array Representation:

In PAR, for each of the nodes contained in k-levels, the following information is stored: a) The parent node number, b) An identifier (0 or 1) to specify if there are other neighboring nodes belonging to the same level (siblings) or previous level (i.e., parent's level), c) If the identifier value is 1, then the number of neighbors identified in the previous step, d) Node numbers for each of the neighbors to identify them. Fig. 7 shows an example of PAR for an arbitrary graph. The node numbers are not stored explicitly but calculated from a value x, which gives the starting node number. As the node numbers are in strictly increasing order, the calculation is simple. Also, neighboring node numbers are not stored explicitly, but calculated from another value y, which is the parent number for x.

The space required is therefore reduced by storing only the differences in the numbering of the parents and neighbors for the nodes, which depends on the value δ (degree of the graph.) While storing the differences between the numbering, the worst case graph, shown in Fig. 8, would give the difference in the order of n. For example, if a level has p nodes, there would be p parent nodes, and say there are other q neighboring nodes. If the node numbers are stored, it would take $((2 \times p + q + p') \times \log_2 n + p)$ bits, where the extra p bits are the identifiers. Also $p' \leq p$ are nodes that have other neighbors and need an additional value indicating the total number of such nodes. Whereas, if the differences in the numbering is used, then it would take in the worst case, $((p + q) \times \log_2 \delta + (2 + p') \times \log_2 n + p)$ bits.

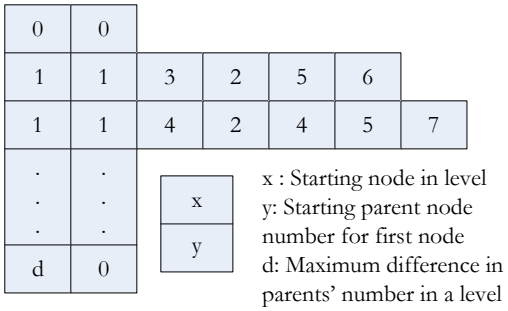


Figure 7: PAR for an arbitrary graph

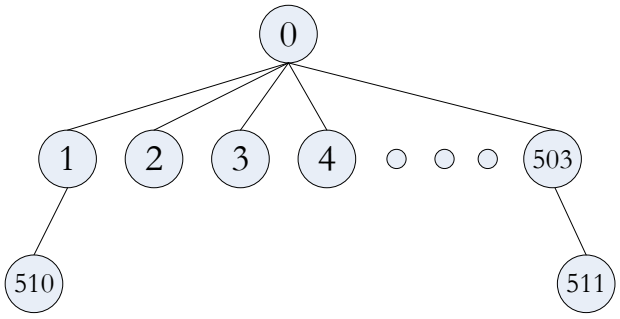


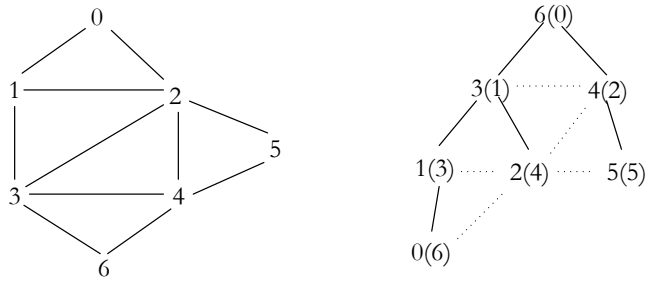
Figure 8: Worst Case: d is order of n

Interestingly, more than one type of storage mechanism can be used depending upon the structure of the BFS-tree. Each of the levels of the BFS-tree would use either *UTM* or *PAR*. The graph is

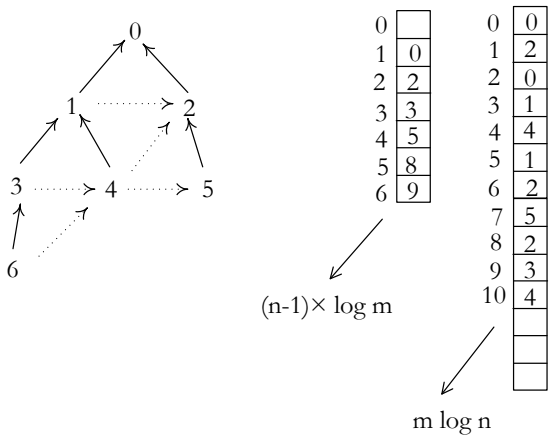
preprocessed in the CPU, and depending on the size of the representations, the smaller data structure is chosen.

6.4.2 Directed BFS-tree and AL-EG Storage

Another data structure that uses breadth-first search tree information and is proposed in this paper is referred to as the directed BFS-tree with adjacency list and edges grouped storage. Here, first the breadth-first search tree is constructed and nodes are provided with BFS numbers. The following links are considered as directed links: a) Child to parent in the BFS tree, b) all non-tree edges from node at level l to nodes at level $l-1$ and, c) all non-tree edges from nodes numbered i to nodes numbered j such that $i < j$ and nodes i and j are at the same level in the BFS tree. This representation is shown with an example in Fig. 9.



An example graph (left) with the corresponding BFS-tree renumbering (right)



Directed-BFS tree (left) with the corresponding adjacency list representation (right) with edges grouped (AL-EG)

Figure 9: Directed-BFS tree with Edges Grouped

For the example graph in Fig. 9, the breadth-first search is performed starting with node number 6. The BFS-tree is shown with the original node numbers along with the BFS numbering in parentheses. Now, only the directed links as defined above are stored in the data structure. So, for the renumbered node 0, there is no information to be stored in the edge array. For the renumbered node 1, there are directed links to nodes 0 and 2, and these are stored in the edge array starting

at location 0. Similarly, for node 2, necessary information is stored in the edge array starting at location 2.

The combinations of nodes to be tested for the counting problems are generated in increasing order of node numbers. So, the above information is sufficient to perform the various required operations on graph data. The main advantage of this data structure over the adjacency list with edges grouped is the data for the edges are stored once instead of twice. For this data structure, the total memory requirement is given by $(n-1) \times \log m + m \times \log n$ bits.

6.5 Comparison of memory requirements for the different data structures using an example:

An example graph of size 16 is provided for illustration. The structure of the original graph and a BFS-tree is shown in Fig. 10. Table 8 compares space requirements for the different data structures.

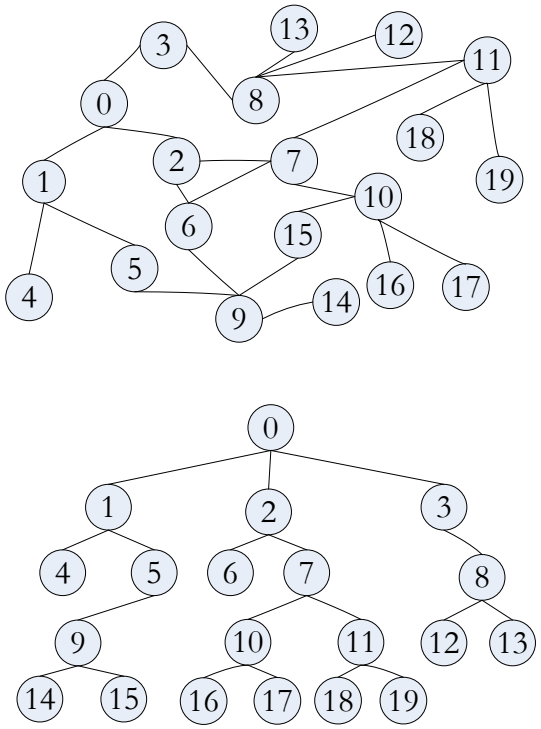


Figure 10: Sample graph (top) and BFS-tree for comparing data structures (bottom)

In case of the S-UTM-ADJ, PAR and the Hybrid data structures, the graph data is split and stored according to level information. For the rest of the data structures, the adjacency information of the entire graph is kept together. Although, from the data in Table 8, it seems S-UTM is the best choice when storing the entire graph together, for larger size sparse graphs other data structures might be more efficient. For example, if for a given graph $n = 300$ and $m = 900$, the corresponding storage requirements in bits for S-UTM is $44,850$; whereas that for the adjacency list with edges grouped is $18,057$ bits and for directed-BFS tree and AL-EG storage is $10,341$ bits.

7 Splitting for larger graphs

In the previous sections, we have seen if the graph fits in the shared memory, we can use one of the several techniques based on BFS-tree. In this section we show how to process graphs where the BFS-tree does not fit in the shared memory.

Table 8: Comparison of space requirements

Data Structure	**Space Required (bits)**
Array of linked lists	4287
Array of arrays	1557
Array of arrays with sizeof()	1471
Array implementation of linked lists	540
Adjacency matrix	400
Adjacency list with edges grouped	300
Upper Triangular Matrix	210
Strictly-UTM	190
Directed-BFS and AL-EG	180
S-UTM-ADJ	174
Parent Array Representation	113
S-UTM-ADJ/PAR: Hybrid	110

We make the following assumption: Given a graph G, there exists a BFS-tree T, such that the graph induced by nodes in any consecutive k levels of T has connected components of size less than 512 in G. Additionally, we assume the entire graph can fit in the shared memory of the 30 streaming multiprocessors. It must be noted that the value 512 corresponds to using S-UTM as the data structure for storing the graph; using other data structures would give different values. However, the underlying principles discussed in this section are applicable to splitting for larger graphs irrespective of the choice of data structures.

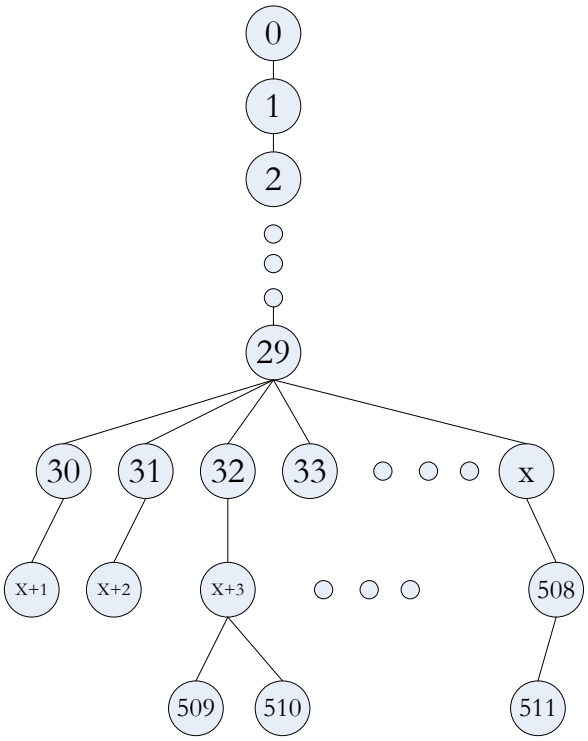


Figure 11: Worst Case BFS-tree for multiple streaming multiprocessors

In the shared memory of each of the consecutive streaming multiprocessors i, nodes in level $k+i$ are added and nodes in level i removed. In this case, if the average number of nodes in each of the levels is l_{avg}, then the total number of nodes that can fit in the shared memory is $512 + l_{avg} \times 29$, where $l_{avg} \leq 512$. This is an improvement over the schemes presented in Section 4 wherein we could process at most 512 nodes even when using the shared memory on all the streaming multiprocessors. In the worst case, if the first 29 levels of T has single nodes, then only 1 new node can be brought in while storing the next consecutive k levels, giving a maximum size graph of $512 + 29 = 541$ nodes. An example BFS-tree of the worst case graph is shown in Fig. 11.

Now, let us assume there exists consecutive k levels of T that do not fit in the shared memory. In this case finding connected components in the graph induced by the nodes in the given k levels might be helpful. As nodes in separate connected components cannot be part of connected subgraphs, calculations can be done on them separately. If there are more than one connected components where each has less than 512 nodes, then calculations can proceed in the following manner; else we have to use the global memory to store them. If any of the components has less than k nodes, then those can be excluded from the calculations. The other components can be kept in the shared memory of a streaming multiprocessor separately, or with other connected components provided the total number of nodes in all does not exceed 512. Now, all possible combinations of size k can be tested from among the nodes of each of the connected components separately.

Algorithm 1: Counting subgraphs of size k using all streaming multiprocessors by splitting G horizontally using T

```
Input: BFS-tree T of graph G
Output: Total count of connected subgraphs of size k
begin
    {L_k} ← divIntoKLevelSets(T);
    {L_k}, {M} ← resetLevlsSMs({L_k}, {M});
    TotalCount ← 0;
    while there are sets of levels marked as new do
        if selected set is the last one then
            while levels available in the set do
                if there are k levels in the memory then
                    Clear memory, mark streaming multiprocessor available;
                else
                    curLvl ← nextAvailableLvl;
                    TotalCount ← TotalCount + testCon(GenNxtComb (curLvl));
                    while there are previous levels do
                        curLvl ← curLvl ∪ prevLvls;
                        TotalCount ← TotalCount + testCon(GenNxtComb (curLvl));
        else
            TotalCount ← TotalCount + testCon(GenNxtComb(fstLvl));
            TotalCount ← TotalCount + testCon(GenNxtComb((allKLvls)));
```

Algorithm 1 takes T as input, and checks for connected subgraphs. The algorithm makes use of all available 30 streaming multiprocessors in the GPU by dividing the work among them. The number of sets of k consecutive levels if there are L levels in T is $Q = L - k + 1$. If $Q > 30$, the remaining sets are brought from the global memory after the current round of operations are completed. With this approach graphs that are stored even in external memories can be processed.

Overview of Algorithm 1: T is divided into sets of k consecutive levels and each is processed by a streaming multiprocessor. The beginning level in the set is processed first, and if it contains more than k nodes, then subgraphs of size k are checked for connectedness from among the nodes in that level. Then all the nodes in the k levels are considered together, and tested for combinations by the function *GenNxtComb(Nodes)* where each combination contains at least one node from the

beginning level and one from the rest of the levels to avoid redundant checking. The above procedure is done for all but the last set of k levels. For the last set, each new level is first processed separately and then combined with all previous levels and checked for combinations provided there is at least one node from both the sets of previous levels and the new level. The function *divIntoKLevelSets(T)* divides T into sets of consecutive k levels, *resetLevlsSMs(Levels, streaming multiprocessors)* resets the streaming multiprocessors marking all of them as available and marks all the k-level sets as new, and ready to be processed. The function *testCon(Comb)* checks if the subgraph induced by the nodes in *Comb* is connected or not.

8 Other related problems

Using similar approaches as adapted in the connectivity testing algorithm, graph problems like finding total number of cliques of size k and total number of independent sets of size k can also be solved. For finding the total number of cliques of size k, only nodes in adjacent levels of T needs to

Algorithm 2: Counting number of k–cliques

```
Input: BFS-tree T of graph G
Output: Total count of cliques of size k
begin
    {L_i} ← divIntoLevels(T);
    {L_i} ← markLevelsNew({L_i});
    TotalCount ← 0;
    TotalLevels ← 0;
    while L_{i.Status} ∈ {L_i} = New do
        curLvl ← L_i;
        TotalCount ← TotalCount + testClique(GenNxtComb (curLvl));
        TotalLevels + +;
        if TotalLevels > 1 then
            curLvl ← L_i ∪ L_{i−1};
            TotalCount ← TotalCount + testClique(GenNxtComb (curLvl));
```

be considered, as given in Algorithm 2. For finding total number of independent sets in G, it's complement say G' is taken, and then a BFS is performed on G' to get T'. Finding cliques of size k in T' is equivalent to finding independent sets of size k in G, as given in Algorithm 3.

Algorithm 3: Counting independent sets of size k

```
Input: Graph G(V, E)
Output: Total count of independent sets of size k
begin
    {G'} ← FindComplement(G);
    {T'} ← BFS-TreeGenerate(G');
    TotalCount ← Algorithm2(T');
```

9 Results

Experiments are performed using both CPU and GPU. The CPU consists of quad-core 2.27 GHz Intel Xeon processors, with 12 GB memory. The GPU used for the experiments is Nvidia C1060 card, with 4 GB device memory and 16 KB shared memory per multiprocessor . The problem of finding the number of connected subgraphs of size k from a graph of size n is solved on the GPU for different values of n and k by storing the adjacency information of the graph in both the shared memory and the global memory, while using both a single streaming multiprocessor and also all available streaming multiprocessors. The number of threads in each of the streaming multiprocessors

considered is limited to 32, which is equal to the *Warp-size* [7]. The data set under consideration is relatively small, comprising of graphs consisting of approximately 300 nodes. Therefore, we do not need too many threads to work on it. If we use more than 32 threads per block, then the number of blocks would decrease and many of the streaming multiprocessors out of the available 30 would go unused. Also, since there are 8 processor cores per streaming multiprocessor, and threads are executed in warps of size 32, keeping the number of threads at 32 per thread block also helps minimize the context switching overhead that may exist.

Fig. 12 plots the timings for evaluating all the combinations for the graph kept on both the shared and global memory while using both single and all available streaming multiprocessors. The plots are as expected, with the timings for the shared memory better than those compared to the global memory. Also, by using all the streaming multiprocessors as compared to using just a single streaming multiprocessor, more threads are available which leads to a better performance.

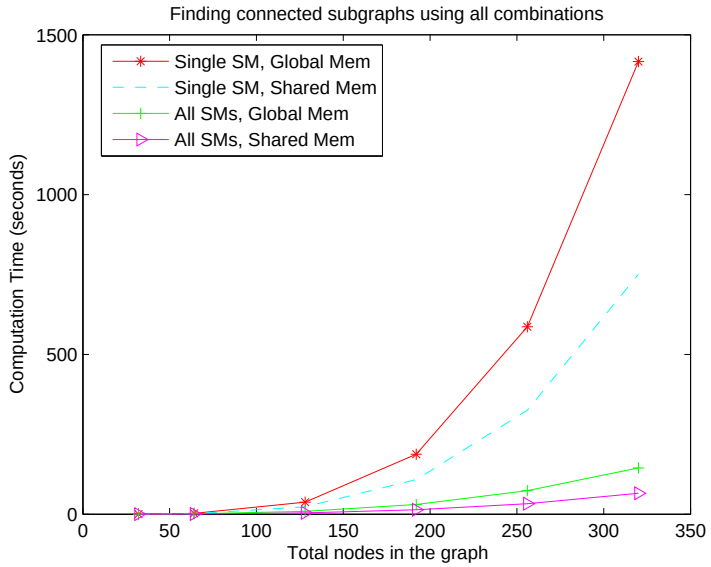


Figure 12: Evaluating all combinations for $k = 3$ with 32 threads in each streaming multiprocessor (SM), for data stored on both shared and global memory

Fig. 13 plots the timings for the graph kept on both the shared and global memory considering the BFS-tree topology information while using both a single streaming multiprocessor and all the available streaming multiprocessors. The number of computations and resulting computation times are greatly reduced in this case as compared to the previous case (Fig. 12) where all the combinations of the nodes are tested.

Fig. 14 plots the timings for the graph kept on the shared memory, using all available streaming multiprocessors, and evaluating both all and reduced number of combinations thereby comparing the previous two cases. It is clear from the Figs. $12 - 14$ that the calculations done using the BFS-tree topology information while keeping the adjacency information on the shared memory and utilizing all available streaming multiprocessors using a large number of threads is the most efficient approach.

The connected subgraph counting problem is also solved on the CPU for subgraphs of size 3. The CPU implementation uses the same data structures and computation techniques as those employed on the GPU. However, the CPU uses a single thread for execution of the program compared to a large number of threads on the GPU. One of the major techniques suggested in the counting problems is to consider the breadth-first search tree information to exclude unnecessary computations. The graph data that is finally used for computation is rearranged according to the BFS-tree levels. Therefore, due to the reordering of the data, the potential for cache line optimization also increases. Hence, the proposed techniques are not only useful for the GPU but also help in optimizing execution on

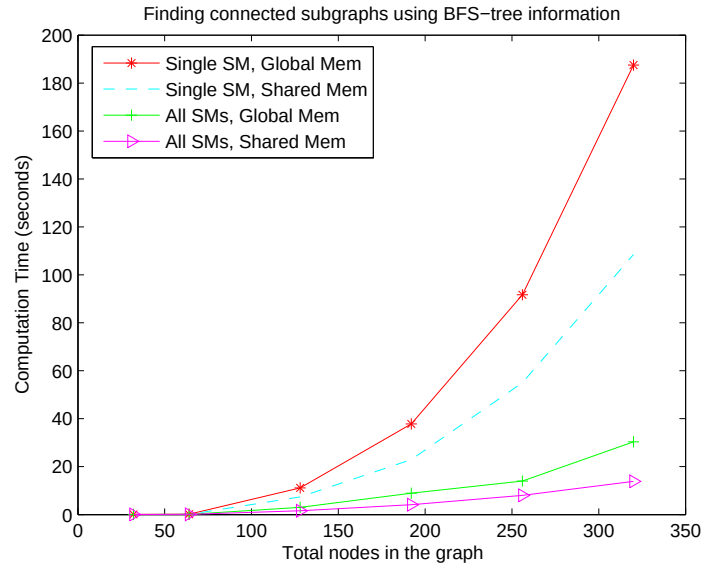


Figure 13: Evaluating reduced number of combinations using BFS-tree information for $k = 3$ with 32 threads in each streaming multiprocessor (SM), for data stored on both shared and global memory

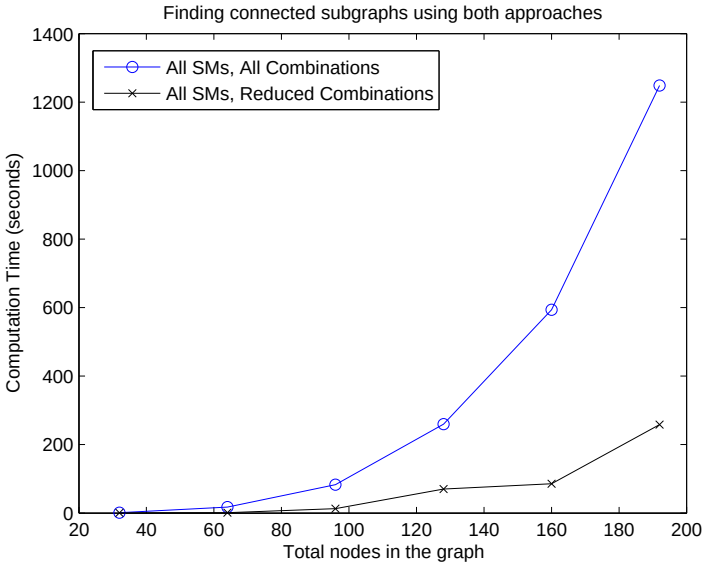


Figure 14: Evaluating all combinations and reduced combinations for $k = 4$ with 32 threads in each of the streaming multiprocessors (SMs), for data stored on shared memory

the CPU too. Fig. 15 provides a comparison of the timings for the implementation on the CPU and GPU. It is evident from the plots that the performance gain when using the GPU to solve the problem is at least $5 - 6$ times as compared to that on the CPU for graphs with more than 300 nodes.

Experiments were also performed using the data available on the Stanford Network Analysis Project [14]. Using reasonably larger graphs of size ranging from $5,000$ to $25,000$ nodes, it can be observed, as shown in Fig. 16, that the computation on the GPU attains a 10 times speedup as compared to that on the CPU. In this case, the graph data is stored on the global memory. This

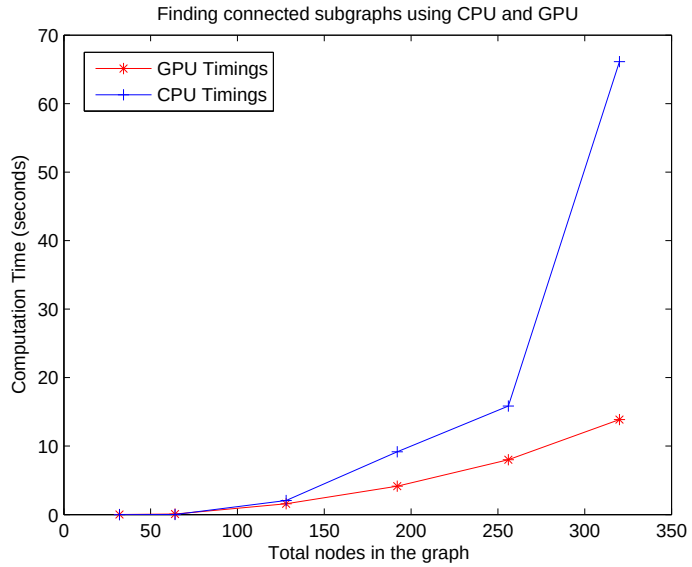


Figure 15: Comparing timings between CPU and GPU for $k = 3$ using BFS-tree information

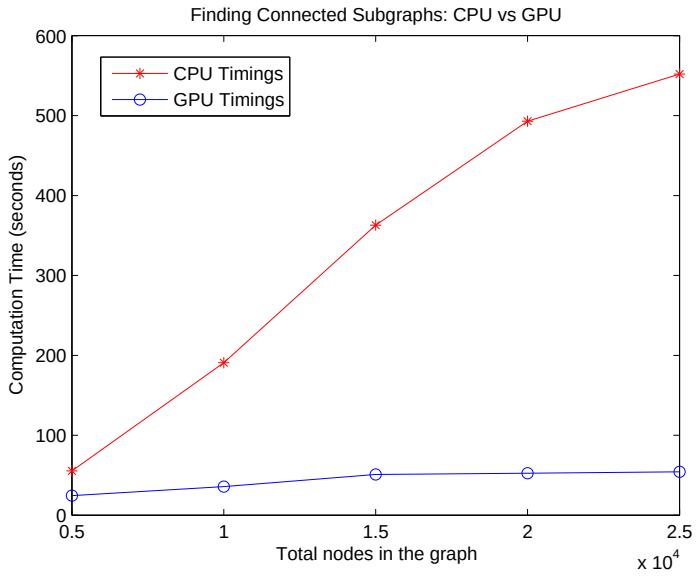


Figure 16: Comparing timings for larger graphs

proves that the proposed solution is applicable for large graph instances too, which is essential for analysis of social networks and other real world datasets.

In the previous experiments, only subgraphs of size 3 and 4 are considered. For larger values of k, the total number of combinations to be checked for specific properties in the graphs induced by the nodes in consideration would increase, provided $k \leq n_k/2$, where n_k is the number of nodes in any set of consecutive k-levels. Since more computation would be required for larger values of k, the speedup is better on the GPU as compared to that on the CPU, as shown in Fig. 17.

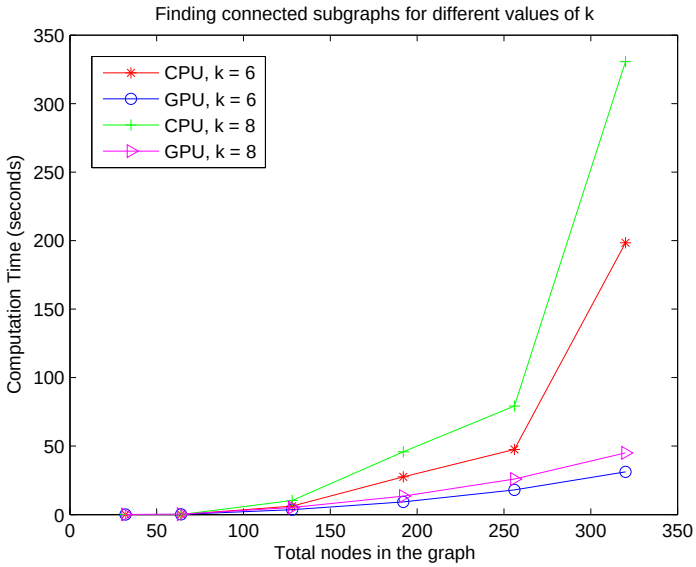


Figure 17: CPU and GPU timings for larger values of k

10 Conclusion

In this paper, algorithms to solve several graph problems on a parallel GPU architecture are discussed. The major focus is on utilizing the faster shared memory of the GPUs and devising data structures to store graphs in the same. Storage requirements and memory access time complexity for various data structures are analyzed in detail. Methods to generate combinations to efficiently divide the work among threads belonging to both single and multiple streaming multiprocessors are developed. In addition, techniques to reduce computations by using breadth-first search tree and exploiting topology information are discussed. Our experimental results show that the smallest computation times are indeed for the graphs stored on the shared memory and calculations using the BFS-tree information. The implementation on the GPU for counting the number of connected subgraphs is faster by a factor of at least 5 as compared to that on the CPU for smaller graphs stored on the shared memory, and by a factor of 10 for larger graphs stored using the global memory. Our future work involves the study of graph compression techniques relevant to the problems under consideration and also using the global memory efficiently for solving graph problems on the GPU.

References

[1] D. A. Bader and K. Madduri. SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1 –12, April 2008.

[2] I. Bordino, D. Donato, A. Gionis, and S. Leonardi. Mining Large Networks with Subgraph Counting. In *Eighth IEEE International Conference on Data Mining*, pages 737 –742, Dec. 2008.

[3] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[4] A. Buluç, J. R. Gilbert, and C. Budak. Solving path problems on the GPU. *Parallel Computing*, 36:241–253, June 2010.

[5] LinkedIn Press Center. http://press.linkedin.com/about, 2011.

[6] A. Chatterjee, S. Radhakrishnan, and J .K. Antonio. Counting Problems on Graphs: GPU Storage and Parallel Computing Techniques. In *IEEE International Symposium on Parallel and Distributed Processing Workshops, APDCM*, 2012.

[7] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, Version 3.2, 2010.

[8] Y. Frishman and A. Tal. Multi-Level Graph Layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13:1310–1319, November 2007.

[9] M. Garland. Sparse matrix computations on manycore GPUs. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 2 –6, June 2008.

[10] F. G. Gustavson, J. Waśniewski, J. J. Dongarra, and J. Langou. Rectangular Full Packed Format for Cholesky's Algorithm: Factorization, Solution and Inversion. *ACM Transactions on Mathematical Software*, 37:18:1–18:21, April 2010.

[11] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the IEEE International Conference on High Performance Computing, LNCS 4873*, pages 197–208, 2007.

[12] T. Itokawa, A. Tada, and M. Migita. Parallel Algorithm for Finding the Minimum Edges to Make a Disconnected Directed Acyclic Graph Strongly Connected. In *Second International Conference on Innovative Computing, Information and Control*, page 131, sept. 2007.

[13] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55. Eurographics Association, 2008.

[14] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[15] Facebook Statistics. https://www.facebook.com/press/ info.php?statistics, 2011.

[16] Twitter Statistics. http://www.geek.com/articles/news/ twitter-reaches-200-million-users-and-110-million-tweets-per-day-20110120, 2011.