# Detecting Cycles in Graphs Using Parallel Capabilities of GPU

Fahad Mahdi[1], Maytham Safar[1], and Khaled Mahdi[2]

[1] Computer Engineering Department
Kuwait University, Kuwait
[2] Chemical Engineering Department
Kuwait University, Kuwait
fahad.a.mahdi@gmail.com, maytham.safar@ku.edu.kw,
khaled.mahdi@ku.edu.kw

**Abstract.** We present an approximation algorithm for detecting the number of cycles in an undirected graph, using CUDA (Compute Unified Device Architecture) technology from NVIDIA and utilizing the massively parallel multi-threaded processor; GPU (Graphics Processing Unit). Although the cycle detection is an NP-complete problem, this work reduces the execution time and the consumption of hardware resources with only a commodity GPU, such that the algorithm makes a substantial difference compared to the serial counterpart. The idea is to convert the cycle detection from adjacency matrix/list view of the graph, applying DFS (Depth First Search) to a mathematical model so that each thread in the GPU will execute a simple computation procedures and a finite number of loops in a polynomial time. The algorithm is composed of two phases, the first phase is to create a unique number of combinations of the cycle length using combinatorial mathematics. The second phase is to approximate the number of swaps (permutations) for each thread to check the possibility of cycle. An experiment was conducted to compare the results of our algorithm with the results of another algorithm based on the Donald Johnson backtracking algorithm.

**Keywords:** graph cycles, GPU programming, CUDA Parallel algorithms.

## 1 Introduction

The communication field is one of the largest fields in engineering, and it is growing faster than any other field. LANs, E-mail, chatting, peer-to-peer networks and even friends websites, all these facilities and many others were created to facilitate the communications between two or more users and appease the user's sociality. Sociality is the most uniqueness of the human being. This uniqueness implied an important question that was considered centuries ago, how do people communicate? What are the rules that really control these communications? Half century ago, it was almost impossible to provide qualitative or quantitative assessment to help providing accurate answers to these questions. However, as the population of the world is growing, the importance of answering this question is increasing [1][2][3].

H. Cherifi, J.M. Zain, and E. El-Qawasmeh (Eds.): DICTAP 2011, Part II, CCIS 167, pp. 193–205, 2011.

Social Network is a graph that represents each actor in the community as a vertex, and the relations between actors as an edge. Social networks are how any community is modeled. The social network model helps the study of the community behavior and thus leveraging social network to researchers demands [1][2] numerous applications of social networks modeling are found in the literature.

An essential characteristic of any network is its resilience to failures or attacks, or what is known as the robustness of a network [9]. The definition of a robust network is rather debatable. One interpretation of a robust network assumes that social links connecting people together can experience dynamic changes, as is the case with many friendship networks such as Facebook, Hi5. Individuals can delete a friend or add a new one without constraints. Other networks have rigid links that are not allowed to experience changes with time such in strong family network. Entropy of a network is proven to be a quantitative measure of its robustness. Therefore, the maximization of networks entropy is equivalent to the optimization of its robustness [9].

In a social network, there are several choices that define the state of the network; one is the number of social links associated with a social actor, known as degree. This definition is commonly used by almost all researchers [9][10][11]. Characterization of a social network through the degree leads to different non-universal forms of distribution. For instance, a random network has a Poisson distribution of the degree. A Small-World network has generalized binomial distribution. Scale-Free networks have a power law distribution form [10]. There is no universality class reported. Instead of nodes degree, the cycle's degree probability [12][13] is a universal distribution form that is applicable for all social networks by using a different definition of the state of the network. Cycles were one of the major concerns in the social network field. In [14], they mentioned that the cycles could be considered as the major aspect that can separate the network to sub-networks or components. Other researches proved that there is a strong relation between the structural balance of a social network and the cycles in the network. Balancing in social network is the set of rules that defines the normal relations between the clients.

The problem of finding cycles in graphs has been of interest to computer science researchers lately due to its challenging time/space complexity. Even though the exhaustive enumeration technique, used by smart algorithms proposed in earlier researches, are restricted to small graphs as the number of loops grows exponentially with the size of the graph. Therefore, it is believed that it is unlikely to find an exact and efficient algorithm for counting cycles.

Counting cycles in a graph is an NP-Complete problem. Therefore, this paper will present an approximated algorithm that counts the number of cycles. Our approximated approach is to design a parallel algorithm utilizing GPU capabilities using NVIDIA CUDA [21] technology.

## 2 Related Work

Existing algorithms for counting the number of cycles in a given graph, are all utilizing the CPU approach.

The algorithm in [15] starts by running DFS (Depth First Search) algorithm on a randomly selected vertex in the graph. During DFS, when discovering an adjacent

vertex to go deeper in the graph if this adjacent vertex is gray colored (i.e. visited before) then this means that a cycle is discovered. The edge between the current vertex (i) and the discovered gray vertex (j) is called a back edge (i,j) and stored in an array to be used later for forming the discovered cycles. When DFS is finished, the algorithm will perform a loop on the array that stores the discovered back edges to form the unique cycles. The cycles will be formed out of discovered back edges by adding to the back edge all edges that form a route from vertex (j) to vertex (i).

The algorithm in [16] is an approximation algorithm that has proven its efficiency in estimating large number of cycles in polynomial time when applied to real world networks. It is based on transferring the cycle count problem into statistical mechanics model to perform the required calculation. The algorithm counts the number of cycles in random, sparse graphs as a function of their length. Although the algorithm has proven its efficiency when it comes to real world networks, the result is not guaranteed for generic graphs.

The algorithm in [17] is based on backtracking with the look ahead technique. It assumes that all vertices are numbered and starts with vertex *s*. The algorithm finds the elementary paths, which start at *s* and contain vertices greater than *s*. The algorithm repeats this operation for all vertices in the graph. The algorithm uses a stack in order to track visited vertices. The advantage of this algorithm is that it guarantees finding an exact solution for the problem. The time complexity of this algorithm is measured as O ((V+E)(C+1)). Where, V is the number of vertices, E is the number of edges and C is the number of cycles. The time bound of the algorithm depends on the number of cycles, which grows exponentially in real life networks.

The algorithm in [18] presented an algorithm based on cycle vector space methods. A vector space that contains all cycles and union of disjoint cycles is formed using the spanning of the graph. Then, vector math operations are applied to find all cycles. This algorithm is slow since it investigates all vectors and only a small portion of them could be cycles.

The algorithm in [19] is DFS-XOR (exclusive OR) based on the fact that small cycles can be joined together to form bigger cycle DFS-XOR algorithm has an advantage over the algorithm in [16] in the sense that it guarantees the correctness of the results for all graph types. The advantage of the DFS-XOR approximation algorithm over the algorithm in [17] is that it is more time efficient when it comes to real life problems of counting cycles in a graph because its complexity is not depending on the factor of number of cycles.

## 3 Overview of GPU and CUDA

GPU (Graphics Processing Unit) [22] [23] [27] is a specialized microprocessor that offloads and accelerates graphics render from the CPU. GPU can perform complex operations much faster than CPU. The design philosophy of GPU is to have massively parallel multi-threaded processor where millions of threads executes on a set of stream processors (minimum of 32 and above) and dedicated device memory (DRAM).

CUDA [22] [23] [26] is an extension to C based on a few easily-learned abstractions for parallel programming, coprocessor offload, and a few corresponding

additions to C syntax. CUDA represents the coprocessor as a device that can run a large number of threads. The threads are managed by representing parallel tasks as kernels (the sequence of work to be done in each thread) mapped over a domain (the set of threads to be invoked). Kernels are scalar and represent the work to be done at a single point in the domain. The kernel is then invoked as a thread at every point in the domain. The parallel threads share memory and synchronize using barriers. Data is prepared for processing on the GPU by copying it to the graphics board's memory. Data transfer is performed using DMA and can take place concurrently with kernel processing. Once written, data on the GPU is persistent unless it is de-allocated or overwritten, remaining available for subsequent kernels

## 4 Thread-Based Cycle Detection Algorithm High Level Design

SPMD (Single Program Multiple Data), an approach that fits well in cases where the same algorithm runs against different sets of data. Cycle count problem can be viewed as systematic view of inspecting all possible paths using different set of vertices. A typical implementation of SPMD is to develop a massively threaded application. The thread based solution of the cycle count can be modeled as algorithm code (SP) plus set of vertices (MD).

Cycle Count Problem for small graph sizes fits well to CUDA programming model. We shall create $N$ threads that can check $N$ possible combinations for a cycle in parallel, provided that there is no inter-thread communication is needed to achieve highest level of parallelism and avoid any kind of thread dependency that degrade the performance of the parallel applications.

The main idea of the thread-based cycle detection algorithm is to convert the nature of the cycle detection problem from adjacency matrix/list view of the graph, applying DFS or any brute force steps on set of vertices to a mathematical (numerical) model so that each thread in the GPU will execute a simple computation procedures and a finite number of loops ($|V|$ bounded) in a polynomial time (thread kernel function time complexity). The algorithm is composed of two phases, the first phase is to create a unique number of combinations of size $C$ (where $C$ is the cycle length) out of a number of vertices $|V|$ using CUDA parallel programming model. Each thread in the GPU device will be assigned to one of the possible combinations. Each thread will create its own set of vertices denoted as combination row (set) by knowing its thread ID. Then each thread examines the cycle existence of combination row vertices to see if they form a cycle or not regardless of the vertices order in the set by using a technique called “virtual adjacency matrix” test. The second phase is to approximate the number of swaps (permutations) for each thread vertices to check other possibilities of cycle occurrence.

The following are the main execution steps of the thread-based cycle detection algorithm, detailed explanation of the steps will be discussed in the next section:

1. Filter all vertices that do not form a cycle.
2. Create a new set of vertex indices in the array called $V'$.
3. For each cycle of length $C$, generate unique combination in parallel.
4. For each generated combination, check cycle existence.
5. Compute the approximation factor ($n$) to be the number of swaps.

6. For each swapped combination, check for cycle existence.
7. After each completion of cycle existence check, decrement ($n$).
8. Send swapped combinations from the GPU memory to CPU memory.
9. Repeat steps (6, 7, 8), until permutation factor ($n$) for all threads equal to 0.

## 5 Thread-Based Cycle Detection Algorithm Detailed Level Design

The following will go through a detailed explanations of the steps listed in section 4. It gives the description of the components that build up the entire thread-based cycle detection algorithm.

### 5.1 Vertex Filtering

Each Element in the $V'$ array contains a vertex that has a degree greater than (2) (excluding self loop edge). The major advantage of filtering "pruning" is to minimize the number of combinations (threads) generated (i.e. combinations that will never form a cycle). Since there are some vertices that do not satisfy the cycle existence, this will create wasteful combinations. For example If $|V| = 10$, looking for a cycle of length $C$=3, then for all vertices the number of combinations to be created without pruning is 240. If we filter the vertices having degree greater than (2); let us say that 3 vertices have degree less than (2), then $|V`| = 7$ (3 out of 10 vertices will be eliminated from being used to generate combinations), then we have to create 35 combinations, results of saving 205 unnecessary combinations.

### 5.2 Vertex Re-indexing

Vertices will be labeled by their index in $V'$ array rather than their value in $V$, for instance If $V$=2 It will be represented as 0 in $V'$. To retrieve the original set of vertices if the set of combinations of $V'$ array forms a cycle, we do a simple re-mapping procedure. For example, if the combination (0, 1, 2, 0) forms a cycle in $V'$ array, then it will be resolved to (2, 3, 4, 2) in $V$ array. The main reason for vertex re-indexing is to allow each thread to generate its own set of combinations correctly, because the combination generation mechanism is relying on an ordered sequenced set of numbers.

### 5.3 Parallel Combination Generator

Given $|V|$, where $V$ is the graph size in terms of the number of vertices, for example if $V = \{1, 2, 3, 4\ldots 10\}$, then $|V| = 10$. Given $|Pos|$ where $Pos$ is the index of Vertex Position to be placed in the cycle length, for a cycle of length $C$, then $|Pos| = C$ , we to have to place vertices in the following order $Pos(1)$, $Pos(2)$, $Pos(3)$,….. $Pos$ $(C)$. For example $V$={1,2,3,4,5,6}, take a cycle of length 4 (2-4-5-6) Then, at $Pos(1)$ will have 2, at $Pos(2)$ will have 4, at $Pos(3)$ will have 5 and at $Pos(4)$ will have 6.

Knowing the first position vertex value of the combination will allow us to know the remaining positions, since the set of vertices that form a given combination are sorted, we can guarantee that at position[i] the value must be at least value of position [i-1] + 1. There is a strong relationship between the row id of the combination and the vertex value of the first position.

Consider an undirected graph of size 6, looking for a cycle of length 3, as an example for showing how the generator works. The target Thread ID is 9, and we want to generate its corresponding entries.

Before we start explaining the algorithm, here is the mathematical expression of each function that is referenced in the algorithm:

$$C_n^m = \frac{m!}{(m-n)! \times n!}.$$

$C_n^m$ is the number of unique subset of size n to be generated from the set m.

$$\mathrm{Vmin}(i) = \mathrm{Pos}(i).$$

Vmin (i) is the minimum possible vertex value to be stored at position (i).

$$\mathrm{Vmax}(i) = V - \big(C - \mathrm{Pos}(i)\big).$$

Vmax (i) is the maximum possible vertex value to be stored at position (i).

$$\mathrm{Offset}\,(k, i) = C_{c-i}^{v-k}.$$

Offset (k,i) is the number of combination rows of given value (k) at position (i).

### 5.4 Virtual Adjacency Matrix

Building the "Virtual Adjacency Matrix" is made by constructing (*C*) by (*C*) matrix without allocating a device memory. Each thread will create its own matrix based on the combination of vertices of size (*C*). The word "virtual" comes from the fact that building the matrix in the code is not a traditional 2 Dimensional array, but it is a nested loop that computes the number of edges within the combination generated by each thread. Since virtual adjacency matrix is implemented as a nested loop, so no memory is required , it gives a yes/no answer for a given set of combinations that they form a cycle or not.

A small example that shows how virtual adjacency matrix works, consider a sample undirected graph and the corresponding actual adjacency matrix representation. Figure 1 shows the sample graph, while figure 2 shows the adjacency matrix representation for the graph in figure 1.

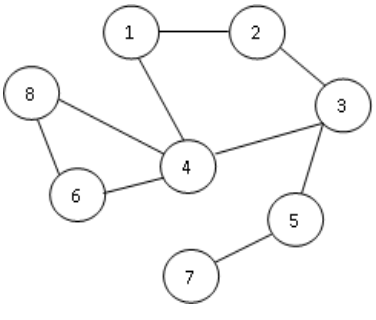


**Fig. 1.** Sample graph used to demonstrate the virtual adjacency matrix test

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 4 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 5 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 8 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

**Fig. 2.** Adjacency matrix representation for the graph in Fig. 1

Case 1: Examine the combination set (1, 2, 3, 4) to see if they form a cycle of length 4 or not, Figure 3 shows the construction of Virtual Adjacency Matrix test:

| | 1 | 2 | 3 | 4 | Edge Count | Passed |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 3-1 = 2 | Yes (1) |
| 2 | 1 | 1 | 1 | 0 | 3-1 = 2 | Yes (1) |
| 3 | 0 | 1 | 1 | 1 | 3-1 = 2 | Yes (1) |
| 4 | 1 | 0 | 1 | 1 | 3-1 = 2 | Yes (1) |

**Fig. 3.** Virtual Adjacency Matrix for the combination (1, 2, 3, 4)

Since $\sum \text{passed} = 4$ which is equal to the Cycle Length, this implies that the combination set (1, 2, 3, 4) forms a cycle (regardless of the vertex order), so the virtual adjacency matrix test will return (TRUE).

Case 2: Examine the combination set (1, 2, 4, 8) to see if they form a cycle of length 4 or not, Figure 4 shows the construction of Virtual Adjacency Matrix test:

| | 1 | 2 | 4 | 8 | Edge Count | Passed |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 3-1 = 2 | Yes (1) |
| 2 | 1 | 1 | 0 | 0 | 2-1 = 1 | No (0) |
| 4 | 1 | 0 | 1 | 1 | 3-1 = 2 | Yes (1) |
| 8 | 0 | 0 | 1 | 1 | 2-1 = 1 | No (0) |

**Fig. 4.** Virtual Adjacency Matrix for the combination (1, 2, 4, 8)

Since $\sum \text{passed} = 2$ which is not equal the Cycle Length, this implies that the combination set (1, 2, 4, 8) do not form a cycle (regardless of the vertex order), so the virtual adjacency matrix test will return (FALSE).

### 5.5 Check for Cycle Using Linear Scan of Edge Count

Once each combination generated passed the virtual adjacency test with (TRUE), a second, detailed cycle detection algorithm will be applied to check if the ordered set of vertices can generate a cycle or not. Because the virtual adjacency matrix test will not tell us what is the exact combination that generates a cycle. The edge count algorithm addresses this issue. The algorithm does a simple nested linear scan for the set of vertices. This algorithm needs temporary array that is assigned to each thread independently called vertex cover of size (*C*). Initially a vertex cover array is initialized to false. A scan pointer starts from the vertex indexed as current passed from the CPU, current vertex will examine its nearest neighborhood to check for an edge connectivity provided that this neighbor is not covered yet, if there is an edge, then a counter called "edge counter" is incremented and flag entry is set to (TRUE) in the covered array for that neighbor vertex is set to true, then setting the newly connected vertex as current. Restart from the first element in the combination array and repeat the procedure again. Keep scanning until the end of the combination array. A cycle exists for the set of vertices if the edge counter is equal to the cycle length, otherwise no cycle is formed.

### 5.6 Swaps of Vertices Set Using Quick Permutation

We have modified an existing permutation of array algorithm called "Quick Permutation reversals on the tail of a linear array without using recursion" [28]. The original algorithm works on sequential behavior on the CPU, we have adopted it to a parallel version that can run on the GPU.

## 6 Approximation Approach Using Permutation Factor (*n*)

The first stage of the thread-based cycle detection algorithm is to create a unique set of combinations of length *C*, then apply check for cycle using virtual adjacency matrix technique, which is yes/no decision make algorithm that gives the answer for the following question: "Is this unique set of vertices (combination) can form a cycle regardless of the vertices order?. We can use the following equation from [20] that is used to compute the maximum number of cycle in an undirected graph:

$$Maximum\ Cycles\ (V, C) = \frac{Perm\ {}^{V}_{C}}{2 \times C} \tag{1}$$

Since $\text{Perm}\ {}^{\text{V}}_{\text{C}} = \text{Comb}\ {}^{\text{V}}_{\text{C}} \times \text{C!}$ , then equation (1) can be express as:

$$Maximum\ Cycles\ (V, C) = Comb\ {}^{V}_{C} \times \frac{C!}{2 \times C} \tag{2}$$

If we do $\frac{C!}{2 \times C}$ permutations for the unique possible combination that are passed the virtual adjacency matrix check with the answer of "yes", then we have covered all

possible cycles, but since $\frac{C!}{2\times C}$ is impractical to execute for even small $C$, say $C$=10, then we need $\frac{10!}{20}$ = 181,440 iterative steps to be executed, which may not be feasible.

Our approximation approach is to minimize $\frac{C!}{2\times C}$ to be a reasonable number, by defining a factor ($n$) that is needed to be multiplied by $\frac{C!}{2\times C}$. The value of ($n$) obtained by the multiplication of the degree percentage of each vertex that is involved in the combination, ($n$) can be expressed as

$$n = \sum_{i=1}^{c}\left(\frac{\text{degree(Vi)}}{\text{Total Degree}}\right) \tag{3}$$

The value of ($n$) will become very small (close to zero) if the percentage degree of each vertex is low, which means that the set of vertices are poorly connected and they have small chances to constitute a cycle. For example if want to do a permutation for a cycle of length 5 with a total degree of 40, the following set of vertices (*v1, v2, v3, v4, v5*) have the degrees (2, 2, 3, 3, 4), respectively. If we use the equation in (3) then:

$$n = \frac{2}{40}+\frac{2}{40}+\frac{3}{40}+\frac{3}{40}+\frac{4}{40} = 0.35$$

If we multiply 0.35 by $\frac{5!}{10}$ will get $n$ = 4, rather than getting 12 (total Permutations). The value of $n$ will become very large (close to one) if the percentage degree of each vertex is high, which means that the set of vertices are strongly connected and they have big chances to constitute a cycle. For example if want to do a permutation for a cycle of length 5 with a total degree of 40, the following set of vertices (*v1, v2, v3, v4, v5*) have the degrees (9, 8, 7, 6, 6) respectively. If use the equation in (3) then:

$$n = \frac{9}{40}+\frac{8}{40}+\frac{7}{40}+\frac{6}{40}+\frac{6}{40} = 0.9$$

If we multiply 0.9 by $\frac{5!}{10}$ will get $n$ = 11, rather than getting 12 (total Permutations).

As a result, based on the connectivity behavior of the vertices either strongly or poorly connected will influence on value the permutation factor ($n$).

Retrieving the value of ($n$) for each possible cycle length, will create an iteration control to determine when to stop permuting.

## 7 Experiments and Results

The experiment in [15] to solve the problem of finding cycles in a graph with 26 nodes used 30 machines in a distributed environment. In our experiments we used only 1 commodity hardware PC, equipped with only 1 nVIDIA 9500 GS GPU device. The cost of 9500 GS GPU card is less than $ 50. The specification of this GPU card is 32 Stream Processors (Core) and 1024 MB Memory (DRAM), all in a single PCI-express graphics card.

Our experiment was conducted on X86-based PC, the CPU is Intel Pentium Dual Core with 2.80GHz. The Memory is 4 GB, equipped with nVIDIA GeForce 9500 GS,

all running under Microsoft windows 7 ultimate edition. The source code written in Microsoft Visual C++ express edition with CUDA SDK v3.2 .

Our experiments showed that the execution time are in seconds. The GPU time is the total execution time to generate combinations kernel function plus check for cycle existence using virtual adjacency matrix.

The CPU time is the total execution time for exporting generated combination stored in GPU memory to the CPU Memory and then dumps the memory content to the secondary storage device. The main reason behind better execution time is that CUDA devices are designed to work with massively parallel thread $2^{41}$.

The next phase of the experiment is to include the permutation factor ($n$) for each cycle length, and do a check for cycle using edge count for each generated permutation. For example if the permutation factor for cycle of length 13 is $n$, we shall check ($n \times 667{,}794$) possible permutations, where each of (667,794) iterations are made in parallel.

We used same graph data in the experiment in [15], and then apply our experiment for a different set of fixed approximation factor ($n$). Table 1 shows the detected cycles and the corresponding execution time for the thread-based cycle detection algorithm.

**Table 1.** Detected cycles and execution time for the approximation solution

| Approximation Factor | Cycles Detected | Execution Time |
|---|---|---|
| 2048 | 20830044 | 3.2 Hours |
| 4096 | 39408671 | 7 Hours |
| 8192 | 76635770 | 18 Hours |

The execution time is the total time used to generate parallel swaps (permutations) by a factor of ($n$) and then check the cycle existence for each swap that is executed at the GPU context plus the total time needed to sum up the total cycles detected at the CPU context.

The experiment in [15] to find the cycles in a graph with 26 nodes takes around 16 days. In our experiment we have solved all the unique possible combination of each cycle length (this is the first phase of the experiment) in less than 2 minutes. Even though in the approximation approach, where for each possible combination that form a cycle we have created ($n$) swaps (this is the second phase of the experiment) it did not last more than 3.2 hours when $n$ = 2048. We can better approximate the solution with more time needed, but still less than the time in the original experiment.

The way of viewing the cycle definition plays an important decision making in the solution feasibility. If the assumption behind solving the problem stated that any cyclic permutation of existing cycle combination considered as one cycle (for instance, cycle 1-2-3-4-1 is the same as -1-3-2-4-1) then applying the first phase of the thread-based cycle detection algorithm (parallel combination generator) will result a significant improvement over existing algorithms. This also achieves time breaker solution compared with other algorithms. But, if the assumption stated that cyclic permutations are considered as individual ones (for instance cycle 1-2-3-4-1 is different as 1-3-2-4-1) then applying approximation approach in the thread-based cycle detection algorithms has a time/resources advantage over other approximation algorithms.

### 7.1 Approximation Accuracy of the Thread-Based Cycle Detection Algorithm

Table 2 shows the approximation accuracy for the number of detected cycles between the exact solution in the algorithm specified in [15] and our thread-based cycle detection algorithm (approximated) alongside with the results obtained from table 2 using the three different approximation parameters ($n$). Also within each run of the approximation factor ($n$), we have included the approximated entropy calculations.

The approximation accuracy measured using the following equation:

$$Approximation\ Accuracy = \frac{Approximated\ Cycles}{Exact\ Cycles} \times 100 \quad (4)$$

**Table 2.** Approximation accuracy (%) of the detected cycles and Entropy calculations for running three different approximation factors ($n$) of the experiment

| Approximation Factor | Accuracy (%) | Entropy |
|---|---|---|
| 2048 | 0.147 | 2.407 |
| 4096 | 0.279 | 2.391 |
| 8192 | 0.542 | 2.355 |

Although the approximation values are quite small, but it can be increased as we increase the permutation factor ($n$), since our main concern in the thread-based cycle detection algorithm design is speeding up the computations.

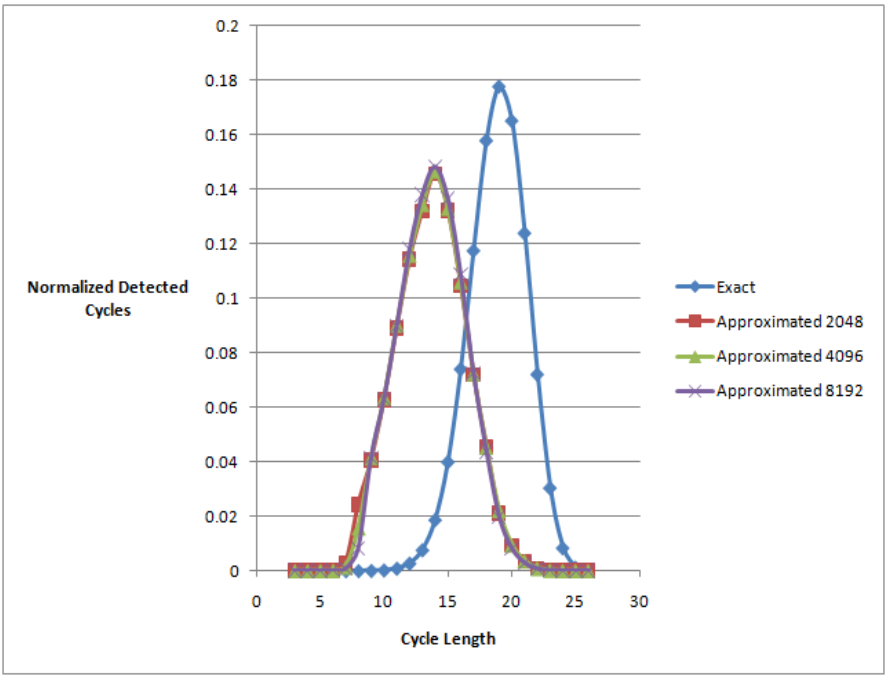


**Fig. 5.** Normalized Results for the exact and approximated solution

Figure 5 shows the normalized results for the exact solution of the detected cycle found in [15] and our approximated solution using three different values of the approximation factor (*n*).

## 8 Conclusions

Cycle count problem is still an NP-Complete, where no such polynomial time solution has been found yet. We have presented a GPU/CUDA thread-based cycle detection algorithm as alternative way to accelerate the computations of the cycle detection. We have utilized GPU/CUDA computing model , since it provides a cost effective system to develop applications that are data-intensive operations and needs high degree of parallelism. The thread-based cycle detection algorithm have viewed the problem from a mathematical perspective, we did not use classical graph operations like DFS. GPU/CUDA works well for mathematical operation, if we can do more mathematical analysis of the problem, we can get better results.

## 9 Future Work

CUDA provides a lot of techniques and best practices that can be applied on the GPU applications to enhance the performance of the execution time of the code. Here we have selected two techniques that are commonly used.

*#pragma unroll* [24] is a compiler directive in the loop code. By default, the compiler unrolls small loops with a known trip count. The *#pragma unroll* directive however can be used to control unrolling of any given loop. It must be placed immediately before the loop and only applies to that loop.

*Shared Memory* [24], because it is on-chip, the shared memory space is much faster than the local and global memory spaces. In fact, for all threads of a warp, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads.

Porting the application to 64 bit platform to support wide range of integer numbers since CUDA provides a 64-bit SDK edition. Converting the idle threads the do not form a cycle to an active threads which form a cycle, by passing such combinations to the idle threads in order to increase the GPU efficiency and utilization .Finding methods to do more filtering procedures for vertices that may not form a cycle, in the current implementation only filter less than degree 2 was used. Migrating the CPU based cycle detect counter by thread procedure to parallel GPU based count , since it creates a performance bottleneck , especially for large number of cycles

## References

1. Shirazi, S.A.J.: Social networking: Orkut, facebook, and gather Blogcritics (2006)
2. Safar, M., Ghaith, H.B.: Friends network. In: IADIS International Conference WWW/Internet, Murcia, Spain (2006)
3. Fiske, A.P.: Human sociality. International Society for the Study of Personal Relationships Bulletin 14(2), 4–9 (1998)

4. Boykin, P., Roychowdhury, V.: Leveraging social networks to fight spam. Computer 38(4), 61–68 (2005)
5. Xu, J., Chen, H.: Criminal network analysis and visualization. Communications of the ACM 48(6), 100–107 (2005)
6. Bagchi, A., Bandyopadhyay, A., Mitra, K.: Design of a data model for social network applications. Journal of Database Management (2006)
7. Bhanu, C., Mitra, S., Bagchi, A., Bandyopadhyay, A.K., Teja: Pre-processing and path normalization of web graph used as a social network. Communicated to the Special Issue on Web Information Retrieval of JDIM (2006)
8. Mitra, S., Bagchi, A., Bandyopadhyay, A.: Complex queries on web graph representing a social network. In: 1st International Conference on Digital Information Management, Bangalore (2006)
9. Wang, B., Tang, H., Guo, C., Xiu, Z.: Entropy optimization of scale-free networks' robustness to random failures. Physica A 363(2), 591–596 (2005)
10. Costa, L.d.F., Rodrigues, F.A., Travieso, G., Villas Boas, P.R.: Characterization of complex networks: A survey of measurements (2006)
11. Albert, R., Barabasi, A.-L.: Statistical mechanics of complex networks. Reviews of Modern Physics 74 (2002)
12. Mahdi, K., Safar, M., Sorkhoh, I.: Entropy of robust social networks. In: IADIS International Conference e-Society, Algarve, Portugal (2008)
13. Mahdi, K.A., Safar, M., Sorkhoh, I., Kassem, A.: Cycle-based versus degree-based classification of social networks. Journal of Digital Information Management 7(6) (2009)
14. Scott, J.: Social Network Analysis: A Handbook. Sage Publication Ltd., Thousand Oaks (2000)
15. Mahdi, K., Farahat, H., Safar, M.: Temporal Evolution of Social Networks in Paltalk. In: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, iiWAS (2008)
16. Marinari, E., Semerjian, G.: On the number of circuits in random graphs. Journal of Statistical Mechanics: Theory and Experiment (2006)
17. Tarjan, R.: Enumaration of the Elementary Circuits of a Directed Graph, Technical Report: TR72-145, Cornell University Ithaca, NY, USA (1972)
18. Liu, H., Wang, J.: A new way to enumerate cycles in a graph. In: International Conference on Internet and Web Applications and Services (2006)
19. Safar, M., Alenzi, K., Albehairy, S.: Counting cycles in an undirected graph using DFS-XOR algorithm, Network Digital Technologies. In: First International Conference on, NDT 2009 (2009)
20. Safar, M., Mahdi, K., Sorkhoh, I.: Maximum entropy of fully connected social network. In: The International Conference on Web Communities (2008)
21. Halfhill, T.R.: Parallel Processing With Cuda Nvidia's High-Performance Computing Platform Uses Massive Multithreading ,Microprocessors Report (January 2008), http://www.MPROnline.com
22. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. The Journal of Parallel and Distributed Computing
23. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)