An Algorithm for Detecting Cycles in Undirected Graphs using CUDA Technology

Maytham Safar, Fahad Mahdi and Khaled Mahdi
Computer Engineering Department, Kuwait University, Kuwait
Research Administration, Kuwait University, Kuwait
Chemical Engineering Department, Kuwait University, Kuwait
maytham.safar@ku.edu.kw, fahad.mahdi@ku.edu.kw, khaled.mahdi@ku.edu.kw

ABSTRACT

Cycles count in a graph is an NP-complete problem. This work minimizes the execution time to solve the problem compared to the other traditional serial, CPU based one. It reduces the hardware resources needed to a single commodity GPU. We developed an algorithm to approximate counting the number of cycles in an undirected graph, by utilizing a modern parallel computing paradigm called CUDA (Compute Unified Device Architecture) from nVIDIA, using the capabilities of the massively parallel multi-threaded specialized processor called Graphics Processing Unit (GPU). The algorithm views the graph from combinatorial perspective rather than the traditional adjacency matrix/list view. The design philosophy of the algorithm shows that each thread will perform simple computation procedures in finite loop iterations to be executed in polynomial time. The algorithm is divided into two stages, the first stage is to extract a unique number of vertices combinations for a given cycle length using combinatorial formulas, and then examine whether given combination can for a cycle or not. The second stage is to approximate the number of exchanges (swaps) between vertices for each thread to check the possibility of cycle existence. An experiment was conducted to compare the results between the proposed algorithm and another distributed serial based algorithm based on the Donald Johnson backtracking algorithm.

KEYWORDS

Approximation algorithms, graph cycles, GPU programming, CUDA parallel algorithms, multi-threaded applications.

1 INTRODUCTION

Complex networks, such as social, biochemical or genetic regulatory networks[1-4] are structures made up of individuals or organizations or chemicals (reactants or products) also referred to as "nodes", which are connected by one or more types of interdependency, such as friendship, common interest, financial exchange or chemical reaction. A known example of a complex networks is a social network, used as a model of a community. The social network model helps predicting community behavior; thus leveraging social network to researchers demands. [1-2] Numerous applications of complex networks modeling are found in the literature; when analyzed, complex network is a graph composed of vertices and edges. [4-5].

An essential characteristic of any network is its resilience to failures or attacks, or what is known as the robustness of a network [6]. The
definition of a robust network is rather debatable. One interpretation of a robust network assumes that 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 [6].

Network robustness is a vital property of complex networks [6-8]. A dynamic system is said to be robust if it is resilient to attacks and random failures. There are several types of threats that a robust network must be secured from. Random vertex removal, an intentional attack to vertices, a network fragmentation and any other event that causes a reduction in the network information-carrying ability can be considered as a threat. In [9], they experimentally found that a scale-free network shows a good resilience to random failures. The heterogeneity of the network degree distribution dictates the chance of randomly attacking a crucial vertex. Depending on this remark, criteria to characterize the complex networks robustness by measuring its heterogeneity were suggested in [6]. They tried to use the principle of entropy to calculate how much the network degree distribution is unbalanced and thus the network heterogeneity. It was proven also that unbalanced degree distribution causes in contrast very low intentional attack survivability [9].

The Entropy of a dynamic network is the number of all configurations (microstates) a network might acquire constrained by its size, nodes and links properties. Statistically describing a network microstate requires defining a microscopic (local) property of the network such as the degree of a node, clusters, cycles or any defined local property. Based on a previous analysis, [6-8] the robustness of a network is best correlated to its ability to deal with internal feedbacks within the network. The more feedback loops (cycles) exist in the network, the more robust the network is. It follows that a fully connected network should represent the most robust networks. Among sets of robust networks, the most stable robust network is a network of size seven because it has the highest cyclic entropy among all sizes. Such interesting result motivates us to explore and search other network topology that has the maximum cyclic entropy.

The topology of an undirected network (graph) can be characterized in term of feedback loops in the network. In other words, an undirected link connecting two nodes is a cycle of degree 2. Extending the definition to three nodes linked together, known as triad, we obtain cycle of degree 3, such definition is a coarse-grain of the former definition, and lesser systems degree of freedom is needed for system representation. The total number of possible configuration using cycle of degree 2 leads to degree entropy. The total number of possible configurations using triads’ representation should lead to triadic entropy of the network. A more generalized definition of network topology is to consider all cycle’s size. The motivation is the inclusion of all
possible feedbacks in the network. The fact that the sums of all data bits in the network are conserved can accurately be represented in a set of cycles than in a set of links or triads. The total number of possible topologies using cycles’ representation leads to the introduction of cyclic entropy, a property of a graph. 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 [14] technology.

2 REALTED WORK

The proposed framework in modeling virtual complex worlds in [21] is using nodes degree level in the graph that represent the virtual world, while the network model was used for the online-communities in video sharing in [28] also using the same concept. Our proposed method is different because we are using the cycles in the network to find out its resiliency. Existing algorithms for counting the number of cycles in a given graph, are all utilizing the CPU approach. The algorithm in [21] 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 [28] 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 [21] 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 [28] 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 [21] 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 [28] 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 [21] 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 GENERAL OVERVIEW OF GPU HARDWARE ARCHITECTURE AND CUDA PARALLEL PROGRAMMING MODEL

The GPU (Graphics Processing Unit) [22][23][29] defined as a specialized microprocessor that offloads and accelerates graphics render from the CPU. As an extension to the existing GPGPU (General Purpose Graphics Processing Unit) systems, 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 execute on a set of stream processors (minimum of 32 and above) and dedicated device memory (DRAM). The GeForce 9500 GS GPU that is used in our implementation is a collection of 4 multiprocessors, with 8 processors results a total 32 core processors. Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches. Each stream processor is connected to the off-chip device memory of 1024 MB in size. At any given cycle, each processor in the multiprocessor executes the same instruction on different data, which makes each a SIMD processor. Communication between multiprocessors is through the device memory, which is available to all the processors of the multiprocessors, figure 1 [29] shows the block diagram of typical GPU structure.

![Figure 1. GPU block diagram.](image-url)
CUDA [22][23][25] 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.

CUDA [23][29] includes C/C++ software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware from developers. CUDA requires programmers to write special code for parallel processing; it doesn’t require them to explicitly manage threads in the conventional sense, which greatly simplifies the programming model. CUDA development tools work alongside a conventional C/C++ compiler, so programmers can mix GPU code with general-purpose code for the host CPU. For now, CUDA aims at data-intensive applications that need single-precision floating-point math (mostly scientific, engineering, and high-performance computing, as well as consumer photo and video editing).

CUDA [26] programming model provides the means for a developer to map a computing problem to such a highly parallel processing architecture. A common design pattern is to decompose the problem into many data-independent sub-problems that can be solved by groups of cooperative parallel threads, referred to in CUDA as thread blocks. Such a two-level parallel decomposition maps naturally to the SIMT architecture: a block virtualizes an SM processor and concurrent threads within the block are scheduled for execution on the SPs of one SM. A single CUDA computation is in fact similar to the SPMD (single-program multiple-data) software model: a scalar sequential program, a kernel, is executed by a set of concurrent threads that constitute a grid of blocks. Overall, a CUDA application is a sequential CPU, host, program that launches kernels on a GPU, device, and specifies the number of blocks and threads per block for each kernel call. Figure 2 shows an abstract view of CUDA programming model [23].
Figure 2. CUDA Programming model.

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. Figure 3 shows conceptual view of the thread-based cycle detection algorithm.

![Figure 3](image)

Figure 3. Conceptual view of the thread-based cycle detection algorithm.

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 DDESIGN

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 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. The table below shows all possible unique combinations for an undirected graph, where \(|V| = 6\) and \(C = 3\). Initially the combination set array for the thread ID (9) is unknown, we shall denoted by \((?, ?, ?)\). Figure 4 shows the possible combination rows for cycles of length 3 for the graph with \(|V|=6\).

![Figure 4. Possible unique combination rows for \(|V|=6\) and \(C=3\).](image)

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 \(m\).

\(V_{\text{min}}(i) = \text{Pos}(i)\).

\(V_{\text{min}}(i)\) is the minimum possible vertex value to be stored at position \((i)\).

\(V_{\text{max}}(i) = V - (C - \text{Pos}(i))\).

\(V_{\text{max}}(i)\) is the maximum possible vertex value to be stored at position \((i)\).

\(\text{Offset}(k, i) = C^2_{n-k}\).

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

Here is the pseudo-code for Parallel Combination Generator:
d_comb : GPU device combination array  
T : Number of combination 
V : Number of vertices 
C : Cycle Length  
CUDA_GENERATE_COMBINATION  
(d_comb_a, T, V, C)  
tid = get_threadID  
if (tid<=T)  
Pos = 1  
k = 1  
Vmax = V - (C - Pos)  
start = 1  
while (TRUE)  
end = start + Cmn (V-k, C-Pos)  
if (tid>= start) AND (tid<= end-1)  
break (while)  
else  
start = end  
k = k + 1  
if (k > Vmax)  
k = Vmax  
break (while)  
d_comb_a[(tid*C)+Pos] = k  
Pos=z  
while (Pos<=C)  
k = d_comb_a[(tid*C)+(Pos-1)]+1  
Vmax = V - (C - Pos)  
while (TRUE)  
offset = Cmn (V-k, C-Pos)  
if (start + offset-1>= tid)  
break (while)  
else  
k = k + 1  
start = start + offset  
if (k >= Vmax)  
k = Vmax  
break (while)  
d_comb_a[(tid*C)+Pos] = k  
pos = pos + 1

The pseudo code is composed of two parts, the first part used to determine the vertex value at position (1), the “while” loop is iterated V times in the worst case. The second part is used to iterate for each remaining positions from position (2) to position (C), in each iteration it will execute “while” loop V times in the worst case. The complexity analysis for executing generate combination procedure is $O(C^2)$, if $C=V$, then $O(V^2)$ where $C$ in $(3, 4, 5, \ldots, V)$.

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 2Dimensional 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 5 shows the sample graph, while figure 6 shows the adjacency matrix representation for the graph in figure 5.

![Figure 5](image1.png)

Figure 5. Sample graph used to demonstrate the virtual adjacency matrix test.
Figure 6. Adjacency matrix representation for the graph in Figure 5.

Case 1: Examine the combination set (1, 2, 3, 4) to see if they form a cycle of length 4 or not. Figure 7 shows the construction of Virtual Adjacency Matrix test:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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 8 shows the construction of Virtual Adjacency Matrix test:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3 = 1 = 2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 8. 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).

The following is the pseudo code for check for cycle existence using Virtual Adjacency Matrix test:

```plaintext
d_cs : GPU device cycle status array
d_adj : GPU device adjacency matrix array
d_comb : GPU device combination array
T : Number of combination
V : Number of vertices
C : Cycle Length
edge_found (x,y): edge status (1 or 0) between x and y
CUDA_CHECK_FOR_CYCLE_1 (d_cs_a,d_adjm_a,d_comb_a,T,V,C)
tid = get thread ID
if (tid<=T)
  passed = 0
  i = 0
  While (i<C)
    x = d_comb_a[(tid*C)+i]
    edge_count = 0
    j = 0
    While (j<C)
      y = d_comb_a[(tid*C)+j]
      edge_count=edge_count+edge_found (x,y)
      j = j + 1
    if (edge_count-1 >=2)
      passed = passed + 1
    i = i + 1
  if (passed = C)
    d_cs_a[tid] = TRUE
else
```

The Society of Digital Information and Wireless Communications, 2012 (ISSN: 2220-9085)
The time complexity analysis for executing the virtual adjacency matrix test operation is \( \mathcal{O}(C^2) \), if \( C=V \), \( \mathcal{O}(V^2) \) then where \( C \) in \((3, 4, 5,\ldots, V)\).

### 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.

The following is the pseudo code to check for cycle existence using linear scan of edge count:

```pseudo
d_cover_a : GPU device vertex cover flag array
d_cs : GPU device cycle status array
d_adj : GPU device adjacency matrix array
d_comb : GPU device combination array
T : Number of combination
V : Number of vertices
C : Cycle Length
N : index of the start vertex to process
edge_found (x,y): edge status (1 or 0) between x and y
CHECK_FOR_CYCLE_2(d_cover_a,d_cs ,d_adj,d_comb,T,V,C,N)
tid = get thread ID
if (tid<=T)
    processed = 0
    edge_count = 0
    d_cover_a[(tid*C)+n] = TRUE
    current = d_perm_a[(tid*C)+n]
    while (processed <= C)
        i = 0
        while (i <= C)
            if (n <> i)
                if edge_found (current,d_perm_a[(tid*C)+i-1])
                    AND d_cover_a[(tid*C)+i] = FALSE
                edge_count = edge_count + 1
                current = d_perm_a[(tid*C)+i]
                d_cover_a[(tid*C)+i] = TRUE
                break (while)
                i =i + 1
            processed = processed + 1
            if (edge_count = C)
                d_cs_a[tid] = TRUE
            else
                d_cs_a[tid] = FALSE
```

The time complexity analysis for executing check for cycle using edge count operation is \( \mathcal{O}(C^2) \), if \( C=V \), then \( \mathcal{O}(V^2) \) where \( C \) in \((3, 4, 5,\ldots, V)\).

### 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” [27]. The original algorithm works on sequential behavior on the CPU; we have adopted it to a parallel version that can run on the GPU.

The following is the pseudo code for the modified version of the quick permutation algorithm:

```
d_pf_a : GPU Device Memory
Permutation Factor Array
d_i_a : GPU Device Memory
Permutation Index Control Array
d_perm_a : GPU Device Memory
Permutation Array
d_index_a : GPU Device Memory
Permutation Index Array
T : Number of Active Combination that can form a cycle
C : Cycle Length
CUDA_PERMUTE_ARRAY()

    tid = get thread ID
    if (tid<=T)
        ax = C - 1
        if (d_pf_a[tid] <> 0)
            i = d_i_a[tid]
            d_index_a[(tid*C)+i] =
            d_index_a[(tid*C)+i] - 1
            i = i - 1
            j = ax
            do
                tmp = d_perm_a[(tid*C)+j]
                d_perm_a[(tid*C)+j] =
                d_perm_a[(tid*C)+i] = tmp
                j = j - 1
                i = i + 1
                while (j > i)
                    i = ax
                while (d_index_a[(tid*C)+i] = 0)
                    d_index_a[(tid*C)+i] = C - i
                    i = i - 1
                    d_i_a[tid] = i
                    d_pf_a[tid] = d_pf_a[tid] - 1
```

The time complexity analysis for permute array is O (C), if C=V, then O (V) where C in (3, 4, 5, ..., V).

6 APPROXIMATION APPROACH USING PERMUATION 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 [28] that is used to compute the maximum number of cycle in an undirected graph:

\[
\text{Maximum Cycles} (V, C) = \frac{\text{Perm}^V_C}{2 \times C}
\]  

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

\[
\text{Maximum Cycles} (V, C) = \text{Comb}^V_C \times \frac{C!}{2 \times C}
\]

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
\[ n = \frac{\sum \text{degree}(v_i)}{\text{Total Degree}} \] (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 \( (v_1, v_2, v_3, v_4, v_5) \) 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 \( (v_1, v_2, v_3, v_4, v_5) \) 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 GPU MEMORY MODEL

The GPU device memory that is used as the primary working area for solving cycle detection problem is basically organized into separated data structures, each structure has its own set of entries so that each thread can access it concurrently. For simplicity, we shall consider that each data structure will be accessed using one Dimensional array configuration. Table 1 shows the memory organization that is used by the algorithm:

<table>
<thead>
<tr>
<th>Area Name</th>
<th>Residency</th>
<th>Size</th>
<th>Thread ID access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency Matrix</td>
<td>CPU/GPU</td>
<td>( V^2 )</td>
<td>N/A</td>
</tr>
<tr>
<td>Filter</td>
<td>CPU/GPU</td>
<td>( V )</td>
<td>N/A</td>
</tr>
<tr>
<td>Vertices Degree</td>
<td>CPU/GPU</td>
<td>( V )</td>
<td>N/A</td>
</tr>
<tr>
<td>Combination (m, n) Values</td>
<td>CPU/GPU</td>
<td>( V )</td>
<td>N/A</td>
</tr>
<tr>
<td>Combination s Work Area</td>
<td>CPU/GPU</td>
<td>( (\text{ThreadID}+C)^+ )</td>
<td>( j )</td>
</tr>
<tr>
<td>Permutations Work Area</td>
<td>CPU/GPU</td>
<td>( (\text{ThreadID}+C)^+ )</td>
<td>( j )</td>
</tr>
<tr>
<td>Vertex Covering Flag Area</td>
<td>GPU</td>
<td>( (\text{ThreadID}+C)^+ )</td>
<td>( j )</td>
</tr>
<tr>
<td>Permutations Index Area</td>
<td>GPU</td>
<td>( (\text{ThreadID}+C)^+ )</td>
<td>( j )</td>
</tr>
<tr>
<td>Permutations Index Control</td>
<td>CPU/GPU</td>
<td>( (\text{ThreadID}+C)^+ )</td>
<td>( j )</td>
</tr>
<tr>
<td>Permutations Factor</td>
<td>CPU/GPU</td>
<td>( (\text{ThreadID}+C)^+ )</td>
<td>( j )</td>
</tr>
<tr>
<td>Cycle Status</td>
<td>CPU/GPU</td>
<td>( V )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\( j \) is the cycle position, have the range from 0 to \( C-1 \)

**Table 1.** GPU memory structure.
• **Adjacency Matrix** is used to store the edge status between the vertices in the graph; it is primarily used by the function: edge_found.

• **Filter Vertices** is an array of a size less than or equal to the total vertices count in the graph, the entries in this array contains the vertices that have degree of (2) and above.

• **Vertices Degree** is an array used to store the total degree value for each vertex.

• **Combination (m, n) Values** is an array used to store the computed value of the combination equation rather than being computed every time a thread needs to compute the Combination of (m, n).

• **Combinations Work Area** is the primary array to be used to store the generated combination by each thread. Generate combination procedure used this array for writing generated vertices.

• **Permutations Work Area** Permutation work area is used to hold temporary vertices values to be used for generating the permutation using the quick algorithm in [27]. Since the GPU programming environment does not support recursion, so we cannot use the recursive version of any permutation algorithm, and therefore we have to replace it with the iterative counterpart.

• **Permutations Index Control** is temporary array used by generate permutation procedure.

• **Vertex Covering Flag Area** is a temporary array used by check for cycle existence using linear scan.

• **Permutations Factor** is a thread based array used to store the approximation factor (n) for each thread.

• **Cycle Status** is a thread based array used to store the cycle existence result for every thread that have either combination or permutation set of vertices.

• **Cycle Count** is cycle length based array used to accumulate the total cycles detected by each thread. Generally, the GPU memory accessed either by each thread individually, where the memory location element is the thread ID this can be seen on Cycle Status array (cs_a). The other way is to indicate the vertex position in the generated/permutated set of vertices, so that the memory accessed is combined between the thread ID and vertex position index. This can be seen on Combination Working Area array (d_comb_a).

The total memory storage of all these data structures should not exceed the maximum memory amount the GPU device can handle, for solving a specific cycle length at a time. For example in a graph size of length 10 we shall reuse the device memory iteratively by interchanging the memory allocation and de-allocation for each cycle length, so for a cycle of length 3 we shall allocated the memory using the formulas above to apply the algorithm, then for a length of 4, we de-allocate the previous memory usage of the and reallocated again and so. We use this technique for optimizing the memory usages as the memory device is fully dedicated to solve the cycle detection of length C at a given time.

**8 CONIDERARTIONS RELATED TO THE THREAD-BASED CYCLE**
DETECTION ALGORITHM DESIGN

Since we are porting the parallelism of the threads from the traditional CPU programming to the GPU, there are some important considerations that must be reviewed first before starting the implementation of the thread-based cycle detection algorithm. These considerations are related to the design limitation of the GPU and the mechanism that it works.

8.1 Maximum Numbers of Combinations to Be Generated Constraint

The maximum number of combinations to be generated for a cycle of length C should not exceed the maximum number of threads supported by CUDA which is \(2^{41}\) according to nVIDIA specifications, as far as the number of possible combinations is less than \(2^{41}\) we can execute the combinations in parallel. But once the numbers of combinations exceed \(2^{41}\), then we need to breakdown the combinations into batches of \(2^{41}\) combinations. In this case, every \(2^{41}\) combinations shall execute in a parallel, passing the results of these combinations to the CPU and start with the next \(2^{41}\) combinations and so on. For example, if the possible combinations are approximately \(2^{50}\), then we have execute \(\frac{2^{50}}{2^{41}} = 2^{9}(=512)\) batches of \(2^{41}\) combinations by CPU iteratively. In our experiment, the largest combinations size was when \(C=13\) for a graph of size 26, which is approximately equal to \(2^{21}\), so we are in the safe margin. As long as the number of combinations is less than the maximum number of threads supported by CUDA \((2^{41})\) or the number of combinations can be approximately to be expressed as \((2^{41})\)multiplied by a factor \((x)\), and that \((x)\) can be a reasonable number, then the approach to solve the problem is still feasible, otherwise it may be not.

8.2 Memory Storage Constraint

Since the thread-based cycle detection algorithm needs a working area for creating combinations and permutations, it shows that the dominating factor for memory consumption is function of \((V!)\) and \((C!)\). These factors will play a critical role in the memory usage as the values of \(V\) and \(C\) increase. In our experiment, \(|V|\) was 26 and peak cycle length was 13, if we substitute the entries in the memory organization table, a total of \(572033754\) memory locations needed. If the unit of memory storage is Byte, then \(\frac{572033754}{1024 \times 1024} = 546\) MB is the total memory used to solve the problem for cycle length 13 for a graph of size 26, In our experiments the GPU device memory specification was 1024 MB. Even though we reduce the time for cycle detection using parallel computing paradigm, the price to be paid is the space complexity. It is extremely difficult to develop an algorithm that can be both time and space efficient.

8.3 CPU Based Export Procedure for the Resulting Data to the Secondary Storage

There is a major performance bottleneck that appears when the GPU would send the detected cycles to the storage. The GPU cannot send directly its memory content to the storage; it must be first transferred to the CPU memory. Then a
sequential loop on the CPU context based on the number of detected cycles will dump the memory content to the storage. If the detected cycles are large (millions) then it will degrade the performance of the entire application. This is because the GPU will become idle waiting for the loop to finish. Because of the serial behavior of the storage access this constraint cannot be avoided.

8.4 The Value of Approximation Factor \((n)\)

The value of \((n)\) plays an influence in the execution time of the thread-based cycle detection algorithm. Although the higher \((n)\) the more accurate results are obtained, but a practical execution time is an issue. A moderate value of \((n)\) is an alternative to be set in order to have a reasonable execution time. Since the goal of the thread-based cycle detection algorithm is to accelerate the computation procedure for the cycle count, the accuracy must be traded off.

9 EXPERIMENTS AND RESULTS

The experiment in [21] 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.

Table 2 shows the number of combinations (threads) generated, CPU, GPU execution time and the detected cycles from the generated combinations for a graph of size 26 starting from a cycle of length 3. Note that the execution time is not including permutations factor iteration.

<table>
<thead>
<tr>
<th>Length</th>
<th>Combinations</th>
<th>GPU Time (sec)</th>
<th>CPU Time (sec)</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2600</td>
<td>0.001</td>
<td>0.000</td>
<td>87</td>
</tr>
<tr>
<td>4</td>
<td>14950</td>
<td>0.004</td>
<td>0.000</td>
<td>361</td>
</tr>
<tr>
<td>5</td>
<td>65780</td>
<td>0.020</td>
<td>0.000</td>
<td>1801</td>
</tr>
<tr>
<td>6</td>
<td>230230</td>
<td>0.091</td>
<td>0.047</td>
<td>7229</td>
</tr>
<tr>
<td>7</td>
<td>657800</td>
<td>0.331</td>
<td>0.172</td>
<td>22586</td>
</tr>
<tr>
<td>8</td>
<td>1562275</td>
<td>0.965</td>
<td>0.485</td>
<td>58520</td>
</tr>
<tr>
<td>9</td>
<td>3124550</td>
<td>2.323</td>
<td>1.156</td>
<td>127802</td>
</tr>
<tr>
<td>10</td>
<td>5311735</td>
<td>4.902</td>
<td>2.328</td>
<td>238465</td>
</tr>
<tr>
<td>11</td>
<td>7726160</td>
<td>8.446</td>
<td>4.047</td>
<td>384954</td>
</tr>
<tr>
<td>12</td>
<td>9657700</td>
<td>11.562</td>
<td>6.063</td>
<td>541841</td>
</tr>
<tr>
<td>13</td>
<td>10400600</td>
<td>15.652</td>
<td>7.969</td>
<td>667864</td>
</tr>
<tr>
<td>14</td>
<td>9657700</td>
<td>15.110</td>
<td>9.187</td>
<td>721221</td>
</tr>
<tr>
<td>15</td>
<td>7726160</td>
<td>13.683</td>
<td>9.141</td>
<td>680863</td>
</tr>
<tr>
<td>16</td>
<td>5311735</td>
<td>11.638</td>
<td>8.031</td>
<td>595943</td>
</tr>
<tr>
<td>17</td>
<td>3124550</td>
<td>7.663</td>
<td>5.922</td>
<td>397939</td>
</tr>
<tr>
<td>18</td>
<td>1562275</td>
<td>4.053</td>
<td>3.468</td>
<td>242986</td>
</tr>
<tr>
<td>19</td>
<td>657800</td>
<td>1.988</td>
<td>1.89</td>
<td>126052</td>
</tr>
<tr>
<td>20</td>
<td>230230</td>
<td>0.765</td>
<td>0.844</td>
<td>54771</td>
</tr>
<tr>
<td>21</td>
<td>65780</td>
<td>0.239</td>
<td>0.312</td>
<td>19553</td>
</tr>
<tr>
<td>22</td>
<td>14950</td>
<td>0.061</td>
<td>0.093</td>
<td>5584</td>
</tr>
<tr>
<td>23</td>
<td>2600</td>
<td>0.012</td>
<td>0.015</td>
<td>1227</td>
</tr>
<tr>
<td>24</td>
<td>325</td>
<td>0.002</td>
<td>0.000</td>
<td>195</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>0.001</td>
<td>0.000</td>
<td>20</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2. GPU, CPU execution time for unique cycle combinations.
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 milliseconds execution time is that CUDA devices are designed to work with massively parallel thread, while in the Table 2 the maximum execution time was for cycle of length 13 (Peak Length), since there around $2^{21}$ combinations generated for this cycle length, which is far away than the maximum number of possible threads generated by the device. 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 [21], and then apply our experiment for a different set of fixed approximation factor ($n$). Table 3 shows the detected cycles and the corresponding execution time for the thread-based cycle detection algorithm.

<table>
<thead>
<tr>
<th>Approximation Factor</th>
<th>Cycles Detected</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>20830044</td>
<td>3.2 Hours</td>
</tr>
<tr>
<td>4096</td>
<td>39408671</td>
<td>7 Hours</td>
</tr>
<tr>
<td>8192</td>
<td>76635770</td>
<td>18 Hours</td>
</tr>
</tbody>
</table>

**Table 3.** Detected cycles and execution time for the approximation solution.

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 [21] 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 as show in table 2. 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.
9.1 Approximation Accuracy of the Thread-Based Cycle Detection Algorithm

Table 4 shows the approximation accuracy for the number of detected cycles between the exact solution in the algorithm specified in [21] and our thread-based cycle detection algorithm (approximated) alongside with the results obtained from table 3 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:

$$\text{Approximation Accuracy} = \frac{\text{Approximated Cycles}}{\text{Exact Cycles}} \times 100$$  \hspace{1cm} (4)

<table>
<thead>
<tr>
<th>Approximation Factor</th>
<th>Accuracy (%)</th>
<th>Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>0.147</td>
<td>2.407</td>
</tr>
<tr>
<td>4096</td>
<td>0.279</td>
<td>2.391</td>
</tr>
<tr>
<td>8192</td>
<td>0.542</td>
<td>2.355</td>
</tr>
</tbody>
</table>

Table 4. Approximation accuracy (%) of the detected cycles and Entropy calculations for running three different approximation factors ($n$) of the experiment.

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.

Figure 9 shows the normalized results for the exact solution of the detected cycle found in [21] and our approximated solution using three different values of the approximation factor ($n$).

10 CONCLUSIONS

Cycle count problem is still 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. Since other algorithms utilize the traditional serial CPU based implementation, we have utilized GPU/CUDA computing model, because it provides a cost effective system to develop applications that are data-intensive operations and needs high degree of parallelism. This is obvious if we look closely to the general solution of problem from a systematic way of exploring the possible occurrence of the cycles by applying the same core algorithm to a different set of vertices. The thread-based cycle detection algorithm have viewed the problem from a mathematical perspective using the equations of combinatorial theory, we did not use classical graph operations.
like DFS (Depth First Search) as other algorithms do. GPU/CUDA fits well for mathematical operations if we can do more mathematical analysis of the problem, we can get better results and the solution will be more closely to the actual one.

11 FUTURE WORKS

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 [29] 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 [29], 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.

Memory Coalescing [29] to maximize global memory bandwidth using the technique of memory coalesce which minimizes the number of bus transactions for memory accesses. Coalescing provides memory transactions are per half-warp (16 threads.) In best cases, one transaction will be issued for a half warp.

Constant Memory [29] the constant memory is a portion from the Device global memory that allows read-only access by the device and provides faster and more parallel data access paths for CUDA kernel execution than the global memory. We can utilize the constant memory by storing the vertex and edges relations (adjacency matrix/list representation of the graph).

Streams and Asynchronous API [29] the default API behavior of CUDA programming model that the Kernel launches are asynchronous function with CPU. CUDA calls block on GPU is serialized by the driver. The introduction of streams objects and asynchronous functions provide asynchronous execution with CPU to give the ability to concurrently execute a kernel and the function. Stream is sequence of operations that execute in order on GPU. Operations from different streams can be interleaved. The kernel function and asynchronous API function from different streams can be overlapped.

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.

12 REFERENCES


