2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

On Analyzing Large Graphs Using GPUs

Amlan Chatterjee, Sridhar Radhakrishnan and John K. Antonio
School of Computer Science
University of Oklahoma
Norman, USA
{amlan, sridhar, antonio}@ou.edu

***Abstract*—Studying properties of graphs is essential to various applications, and recent growth of online social networks has spurred interests in analyzing their structures using Graphical Processing Units (GPUs). Utilizing the faster available shared memory on GPUs have provided tremendous speed-up for solving many general-purpose problems. However, when data required for processing is large and needs to be stored in the global memory instead of the shared memory, simultaneous memory accesses by threads in execution becomes the bottleneck for achieving higher throughput. In this paper, for storing large graphs, we propose and evaluate techniques to efficiently utilize the different levels of the memory hierarchy of GPUs, with the focus being on the larger global memory. Given a graph G = (V, E), we provide an algorithm to count the number of triangles in G, while storing the adjacency information on the global memory. Our computation techniques and data structure for retrieving the adjacency information is derived from processing the breadth-first-search tree of the input graph. Also, techniques to generate combinations of nodes for testing the properties of graphs induced by the same are discussed in detail. Our methods can be extended to solve other combinatorial counting problems on graphs, such as finding the number of connected subgraphs of size k, number of cliques (resp. independent sets) of size k, and related problems for large data sets. In the context of the triangle counting algorithm, we analyze and utilize primitives such as memory access coalescing and avoiding partition camping that offset the increase in access latency of using a slower but larger global memory. Our experimental results for the GPU implementation show at least 10 times speedup for triangle counting over the CPU counterpart. Another 6–8 % increase in performance is obtained by utilizing the above mentioned primitives as compared to the naïve implementation of the program on the GPU.**

***Index Terms*—Triangle counting, CUDA, graph problems, global memory, GPU optimization.**

I. Introduction

Solving general-purpose problems on the Graphical Processing Units (GPUs) is a highly efficient and low-cost alternative to currently available multicore Central Processing Units (CPUs). Available hardware and the potential for massive multi-threading has shifted the focus from using a GPU as a graphics renderer to a powerful co-processor. Among other problems that have achieved speedup from being solved on the GPU, analysis of graphs, which is an inherent operation in many real world applications, with combinatorially explosive number of computations has the potential to exploit the available architecture of GPUs.

In the past decade there has been tremendous growth in the World Wide Web and Online Social Networks (OSNs) [15] [14], which have provided huge amounts of data in the form of graphs, and their properties have been studied by both academic and commercial organizations. Understanding the structure of OSNs help in improving Internet search, advertising, and even mitigating against spamming and other security issues like Sybil attack [19]. Finding subgraphs that satisfy a specific property in such networks, and solving other problems based on locality information is therefore of practical significance. Although there are a large number of processors available on the GPUs, size of the shared memory, which has the least latency and parallel access, is limited. Most of the data available for online social networks, and the properties of the graphs that define the underlying structure, point towards large connected components in the graph. Therefore, to compute on these large data sets, using the larger but slower global memory is the most efficient option.

In this paper, we focus on methods to optimally utilize the global memory to solve graph problems for large data sets. Given a graph $G = (V, E)$, we provide an algorithm to count the number of triangles in G, while storing the adjacency information on the global memory. We analyze the patterns of global memory access and discuss methodologies to utilize primitives such as memory access coalescing and avoiding partition camping that offset the penalties of using a slower but larger memory in the context of the triangle counting algorithm. Therefore, our methods help to solve problems on large data sets, that are no longer limited to the size of the smaller shared memory, but makes use of the much larger global memory. However, these techniques can also be applied to solve other problems [5] for larger data sets.

The outline of our paper is as follows. In Section II, we present information on previous work related to solving large graph problems and optimization techniques for increasing the efficiency of memory access while using the global memory. In Section III, background information on computing on the shared memory of the GPU is provided. Section IV discusses the memory hierarchy and storage of graphs on the GPU. Techniques for handling larger graphs using breadth-first search tree information are analyzed in Section V. In Section VI, scheduling computations on available GPU cores is studied, and the problem is found to be NP-hard. An algorithm for counting the number of triangles in graphs is given in Section VII. Section VIII deals with the techniques to efficiently generate combinations for testing properties in graphs. Section IX discusses the memory coalescing technique both in general

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.235

and in the context of the triangle counting algorithm. Section X analyzes the avoidance of partition camping primitive that helps to improve the usage of the global memory. For triangle counting, methods for data re-organization in the memory are provided and analyzed. Results of the implementation of the triangle counting algorithm on both CPU and GPU are compared in Section XI, along with the added improvements by using the above mentioned primitives. Conclusion and future work is discussed in Section XII.

II. Related work

Different data structures and modifications, combined with various optimization techniques are considered for efficiently solving problems on graphs. The algorithm described in [10] by Katz and Kider handles graph sizes that are inherently larger than the DRAM memory available on the GPU by dividing into smaller blocks. The shared memory cache efficient algorithm to solve transitive closure and all-pairs shortest path problem on the GPU in [10] deals with directed graphs for large data sets. However, it does not explicitly propose techniques to use the global memory which would be essential to store such data.

Vineet et al. [17] study an algorithm for fast minimum spanning tree for large graphs on the GPU. The authors in [17] focus on the algorithmic aspect, rather than overcoming the hardware limitations encountered while using the global memory as the storage.

Buluc et al. [4] look at an alternative way of storing the graphs on the GPU by dividing the adjacency matrix into smaller blocks. These representations, although efficient in terms of size of the data structure, might not be ideal for data stored on the global memory, to be operated by a large number of threads as available on the GPUs, causing memory accesses to become sequential. Data structures with redundant information might be more suited to efficiently utilize the larger global memory by avoiding sequential data accesses.

Harish and Narayanan [8] describe methods to accelerate large graph algorithms on the GPU using CUDA. Although the algorithms are implemented in the global memory using compacted adjacency list, there is no discussion on using available primitives to offset the usage of the slower memory.

Hong et al. [9] discuss techniques to improve the work balance among active threads in the level of warps for maximizing the efficiency for solving graph algorithms on CUDA. The methods suggested in [9] study the trade-off between achieving load balancing and utilization of the GPU cores. But, there is a lot of fine-tuning involved and the parameters for the optimum case are dependent on the structure of the input graph.

Ruetsch and Micikevicius [13] have proposed techniques to use available primitives in the context of matrix transpose problem. The results show that tuning the data structures depending on the application can yield improved bandwidth for the data access from the global memory which is comparable to that of the shared memory. Our partition camping avoidance techniques are based on similar principles. However, the data structures used in our research and the computations are entirely different based on the additional information that we incorporate by studying the breadth-first search tree properties of the input graph.

Yang et al. [18] provide general methods for constructing an optimization compiler for GPUs. In [18], the authors study and analyze the basics of identifying the scope for efficient data access from the global memory in the context of matrix multiplication. The main focus in [18] is on the automatic transfer of the code by the optimizing compiler using various other methodologies, including the usage of shared memory and re-organizing the thread blocks for efficient code generation.

Chatterjee et al. [5] describe techniques to solve graph problems on the GPU using only the shared memory. Extending the work from [5], in this paper, we propose an algorithm for triangle counting on large graphs, with the objective of storing the data in the global memory of the GPUs. Counting triangles have many applications including the analysis of social networks as described in [16] [1]. Simple data structures, including those storing redundant information are considered to take advantage of the available memory access optimizers like using memory coalescing and avoiding partition camping.

III. Using shared memory to solve graph problems

In our previous work, we describe techniques to solve counting problems on graphs using GPUs [5]. Specifically, we provided algorithms to count the number of connected subgraphs of size k, the number of independent sets of size k and also the number of k-cliques for a given graph $G = (V, E)$, where $|V| = n$.

Instead of testing for all possible combinations of nodes using a brute-force approach, which is combinatorially explosive with a value of ${}^nC_k$, we show that utilizing the breadth-first search tree information of the input graph significantly reduces the computation overhead by considering nodes only in k adjacent levels in the BFS-tree T of G.

Now, to efficiently solve the problems using the streaming multiprocessors available on the GPU, the data was restricted within the shared memory to take advantage of the least memory latency. However, even while using efficient data structures, the maximum size of the graph that can be computed on is 512. Taking advantage of the BFS-tree information, the graph can be split horizontally into sets of k adjacent levels. This technique increases the maximum size of the graph to 541 nodes in the worst case.

In this paper, we compute on graphs with approximately $100,000$ nodes. Therefore, we propose techniques to efficiently use the global memory on the GPU for accessing adjacency information store on it. Also, naïve combination generation techniques were used to test for the properties in the graph, which are computationally intensive and also resulted in uneven distribution of computation load on the available streaming multiprocessors.

IV. Storing Graphs on GPUs

Adjacency data of the input graph required for computation is stored on the GPU. Memory hierarchy of the GPU consists of global memory, shared memory, constant memory, texture memory and registers. The size of each of the above mentioned types of memory varies according to the system, and a comparison is given in Table I. The global memory is the largest and also has the highest access latency. The on-chip shared memory, which is further divided into 16 (or 32) banks, has significantly faster access compared to the global memory. But, when data is accessed from the same bank, significant performance loss occurs due to bank conflicts (the only exception being the case where all the threads access the same element leading to a broadcast.)

TABLE I
Architecture Comparison of Different Nvidia GPUs

Model #	Cores	Global Mem. (GB)	Sh. Mem. (KB)	# of Mem. Banks	Comp. Cap.
C1060	240	4	16	16	1.3
C2050	448	3	48	32	2.0
C2070	448	6	48	32	2.0

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. Now, to fit the graph in memory, the size required must be less than or equal to the space available. Therefore, for storing graphs using adjacency matrix data, the following equation must be satisfied

$$n^2 \leq S_{mem} \tag{1}$$

where, S_{mem} is the size of the memory in bits.

For undirected graphs, values (i, j) and (j, i) are same. So, storing only the Upper Triangular Matrix (UTM) of the adjacency matrix is enough. Therefore, in this case,

$$\frac{n \times (n+1)}{2} \leq S_{mem} \tag{2}$$

As all the values of (i, j) = 0 when $i = j$, using the Strictly UTM representation (S-UTM) (i.e. without the data on the diagonal), size of the largest graph increases by 1. Using Equation (1) and Equation (2), the largest graph that can be kept in the shared memory and global memory for the different systems is given in Table II.

TABLE II
Maximum size of graphs on different GPUs

Model #	Shared Mem. Adj Mat	Shared Mem. S-UTM	Global Mem. Adj. Mat	Global Mem. S-UTM
C1060	362	512	185,363	262,144
C2050	627	887	160,529	227,023
C2070	627	887	227,023	321,060

V. Handling Larger Graphs

For graphs that fit in the shared memory, algorithms described in [5] can be used to do the computations. In this Section we consider graphs of larger sizes. The following assumptions can be made for the properties of such graphs:

- For a given graph $G = (V, E)$, the adjacency information for G does not fit in the shared memory i.e.,

  $$S_G \geq S_{SM} \tag{3}$$

  where, S_G is the size of G in bits using the most efficient data structure, and S_{SM} is the size of the shared memory in bits.
- G can be preprocessed on the CPU and split into chunks taking into consideration consecutive levels of BFS-tree T of G i.e.,

  $$G \Rightarrow G_1 \cup G_2 \cup \cdots \cup G_i \tag{4}$$

- Let $G = (V, E)$, where $V = \{v_1, v_2, \cdots, v_n\}$. Starting with node v_i, G can be split into subgraphs, where $G = G_{i1} \cup G_{i2} \cup \cdots \cup G_{ij}$. Let, $S = \{s_1, s_2, \cdots, s_n\}$, where

  $$s_i = \sum_{m=1}^{\alpha_i} C_{im} \tag{5}$$

  where, $C_{im} = 1$ if $S_{G_{im}} > S_{SM}$, else $C_{im} = 0$, and α_i is the number of splits possible starting with node v_i. Therefore, to ensure the minimum number of chunks with size greater than the shared memory is chosen s_i is selected, where $s_i = \text{Min}\{\text{S}\}$, and the output of the preprocessing is G_{i1}, G_{i2}, $\cdots$, G_{im}.
- To ensure that the fragmentation is the least i.e., minimum wastage of shared memory (for the chunks that actually fit in the shared memory), the following condition must be satisfied

  $$Min\{S_{SM} \times P - \textstyle\sum_{m=1}^{\alpha_i} S_{G_{im}}, \forall S_{G_{im}} \leq S_{SM}\}$$

  where, P is the number of streaming multiprocessors or modules in the Nvidia system concerned.

Algorithm 1 splits input graph G into sets of consecutive level nodes using Breadth-first search property and considering each connected component of G separately.

After splitting the graph using Algorithm 1, certain sets of nodes or chunks might not fit in the shared memory. The adjacency information for the nodes in such sets is kept in the global memory. Therefore, now the threads in the GPU access data from both shared and global memory.

Due to the difference in access latency of global and shared memory, threads operating on data stored in the global memory might require significantly more time to do the computations. Therefore, if the computations on the data stored in the shared memory and the global memory are performed sequentially, it would take more time than an intelligent scheduling of the computations on the streaming multiprocessors so that, at any instant of time during execution, active warps operate on adjacency data stored in both shared and global memory.

Algorithm 1: Splitting G on the CPU

Input: Graph G
Output: Graphs G_1,G_2,$\cdots$, G_i,
where $G = G_1 \cup G_2 \cup \cdots \cup G_i$
begin
  $\{CC_i\} \leftarrow$ `findConnectedComponents`$\{G\}$;
  foreach CC_i **do**
    while $CC_{i\_size} \geq S_{SM}$ & $v_i \in CC_i(V)$, *where* $\exists\ v_i \notin$ *processed* **do**
      $\{T_i\} \leftarrow$ `createBFStree` $\{CC_i, v_i\}$;
      $\{L_i\} \leftarrow$ `divIntoConsLevelSets`(T_i);
      if $L_{i\_size} \leq S_{SM} \forall L_i$ **then**
        Output $\leftarrow L_1, L_2, \cdots, L_i$;
        Mark CC_i processed;
        `break`;
      Mark v_i processed;
    if $CC_i \notin$ *processed* **then**
      Output $\leftarrow CC_i$;

However, it might be the case for specific instances of graphs, where none of the sets of nodes or chunks fit in the shared memory, and all the sets are in the global memory. In that case, operations on the data can be performed accessing the global memory efficiently taking advantage of memory coalescing and avoiding partition camping as discussed in Section IX and Section X respectively.

The above idea can be illustrated using the following example. Let the total number of sets of data to be computed on be ψ. Also, let the number of sets of data that fit in the shared memory be given by ψ_s and that in the global memory be ψ_g, where $\psi_s + \psi_g = \psi$. Suppose it takes on an average τ_s units of time to operate on a set of data stored in shared memory and τ_g be the corresponding value for data in global memory. If $\psi_s \leq 30$ and the operations on the data stored in shared memory and that in the global memory are performed one after the other as mentioned above, the total time taken for execution, is given by $\tau_s + \psi_g \times \tau_g$, since the data in the shared memory is accessed in parallel and that in the global memory is accessed in a sequential manner. Therefore, for the general case, the total time taken to compute on all the sets of data is given by the following equation:

$$\tau_t = \mu \times \tau_s + \psi_g \times \tau_g \tag{6}$$

where, τ_t is the total time and $\mu = \left\lceil \frac{\psi_s}{30} \right\rceil$. An efficient scheduling of the active warps would minimize the value given by Equation (6).

VI. Scheduling threads to operate on data chunks: Makespan Scheduling

After the given graph G is split into chunks by using Algorithm 1, the data corresponding to these are stored either in the shared or global memory, as required. Now, blocks of threads executing on the streaming multi-processors are scheduled to operate on the data. The objective is to minimize the total time of execution as discussed in the previous Section. This can be done by scheduling the operation on the data for each of the chunks on the available GPU streaming multiprocessors, so that the time required is minimum. This problem is equivalent to the Makespan Scheduling problem, and is NP-hard [7].

The Makespan Scheduling problem is defined as follows: Given Θ machines for scheduling, indexed by the set $M = \{1, \cdots, \Theta\}$, and η jobs, indexed by the set $J = \{1, \cdots, \eta\}$, where job j takes $p_{i,j}$ units of time if scheduled on machine i, and J_i is the set of jobs scheduled on machine i. Then $l_i = \sum_{j \in J_i} p_{i,j}$ is the load of machine i. The maximum load $l_{max} = \max_{i \in M} l_i$ is called the makespan of the schedule.

In the context of the computations on the GPU, the machines are the modules i.e., streaming multiprocessors, and all modules are identical, and the processing time of the jobs are the size of the chunks. Therefore, the value of $p_{i,j}$ and $p_{k,j}$ are same. But, the problem is still NP-hard even if there are only two identical machines, which is much simpler than the problem at hand with 30 identical streaming multiprocessors.

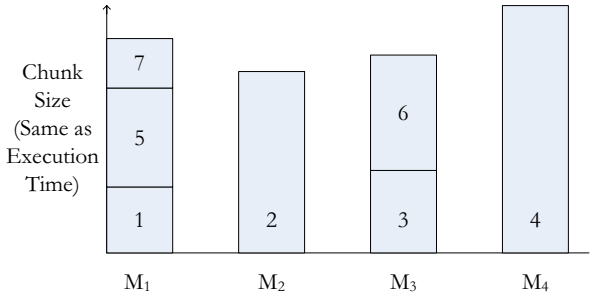

Fig. 1. Executing chunks on GPU cores: Makespan scheduling

An example of the Makespan Scheduling for the given problem is illustrated using the diagram in Fig. 1. Here, M_i's represent the machines i.e., the streaming multiprocessors (just 4 are shown for simplicity), and the rectangles numbered from 1 through 7 represent the jobs i.e., computations to be performed on the chunks. Threads in each module operates on the chunks and the time required is proportional to the size of the chunks. In this case, while chunks 1, 5 and 7 are computed sequentially by module M_1, module M_2 operates on chunk 2, module M_3 operates on chunks 3 and 6 and module M_4 operates on chunk 4, all in parallel. However, as mentioned earlier, assigning chunks to the modules so as to minimize the maximum makespan is NP-hard.

VII. Counting Triangles in Graphs

Finding and counting triangles in graphs is a fundamental problem, and can be solved using GPUs. It is a common graph-mining task, as the number of triangles is closely connected with estimating the clustering coefficients and the transitivity ratio of the graph. Counting the number of triangles has other

applications too, such as spam detection [1]. In the context of social networks, triangles have a natural interpretation: friends of friends tend to be friends, and this can be used in potential friend suggestion, as shown in Fig. 2. Counting triangles in large graphs is described in [1].

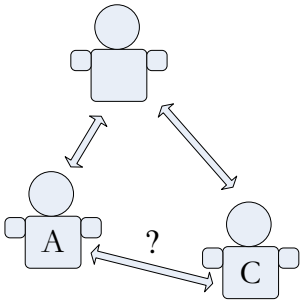

Fig. 2. Triangles in Online Social Networks

A triangle $\Delta = (V_\Delta, E_\Delta)$ of a graph $G = (V, E)$ is a three node subgraph with $V_\Delta = \{u, v, w\} \subseteq V$ and $E_\Delta = \{u, v\}, \{v, w\}, \{w, u\} \subseteq E$. Let $\vartheta(G)$ denote the number of triangles in graph G. It can be noted that an *n*-clique has exactly ${}^nC_3$ triangles i.e., $\vartheta(n\text{-clique}) = {}^nC_3$.

Operations on graphs for finding triangles can be of two types: a) *counting*: finding the number of triangles, and b) *listing*: identifying the nodes forming the triangle, and reporting the same. In this section we discuss the algorithm and implementation methodologies for *counting* triangles in graphs. Optimization techniques that help in increasing the efficiency are discussed in the later Sections. Algorithm 2 finds triangles in a given graph G. It can be noted that the function *GenNxtComb*() finds combinations of 3 nodes for further testing. When *GenNxtComb*() is called with the parameter $bothLvls$, it returns combinations containing 3 nodes from the set of consecutive levels, out of which at least 1 is from the $firstLvl$. This restriction eliminates duplicate checking for any combination of nodes. Fig. 3 shows the grouping of adjacent levels of a sample BFS-tree required for counting the number of triangles.

Apart from counting the number of triangles that exist in a given graph, Algorithm 2 can also check if a graph is *triangle-free*. A triangle-free graph is an undirected graph in which no three vertices form a triangle of edges. Triangle-free graphs are equivalent to graphs with *clique number* ≤ 2, or graphs with *girth* ≥ 4.

VIII. Generating combinations for testing in graphs

The problem we are trying to solve is to verify certain properties in subgraphs of a specific size k for a given graph $G = (V, E)$, where $|V| = n$. Examples of such properties include connectivity, formation of cliques and formation of independent sets. To solve the above mentioned problem, sets of k nodes from n available nodes are to be chosen, and then verified for the required property. This leads to generation of combinations of all possible sets of nodes to check for the correct result. Following are some of the approaches that can be adopted to achieve the desired outcome.

Algorithm 2: Counting number of triangles in G

```
Input: BFS-tree T of graph G
Output: Total number of triangles in G
begin
    {L_i} ← divIntoConsecutiveLvlSets(T);
    TotalCount ← 0;
    foreach L_i do
        TotalCount ← TotalCount +
          testTriangle(GenNxtComb(firstLvl));
        TotalCount ← TotalCount +
          testTriangle(GenNxtComb(bothLvls));
        if L_i is the last set then
            TotalCount ← TotalCount +
              testTriangle(GenNxtComb
              (SecondLvl));
    Output ← TotalCount;
```

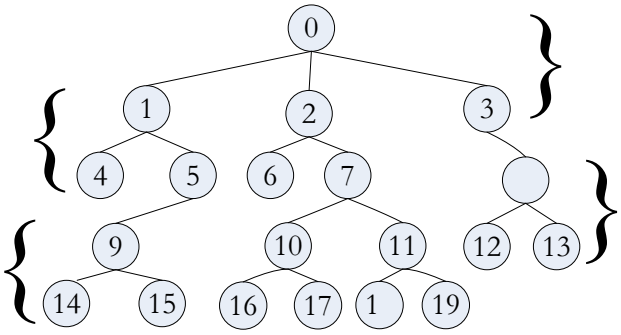

Fig. 3. BFS-tree level grouping for finding triangles

A. Naïve sequential approach with pre-computed combinations

A basic method to test for all combinations is to generate the same in the pre-processing stage and store them. During the actual computation, the combinations can be retrieved and tested for the desired property. The major drawback with this approach is the amount of memory required just to store the pre-computed combinations. For a graph with n nodes and for subgraphs of size k, there would be ${}^nC_k$ tests in total. Now, each of the combinations consist of k values, each of which required $log(n)$ bits for storage. Therefore, the total storage required for such an instance is ${}^nC_k \times k \times log(n)$ bits.

B. Naïve sequential approach with combinations generated on the fly

The previous approach can be improved by not storing the combinations. During the testing phase, the combinations can be generated one-by-one sequentially according to the lexicographical order [12]. This method requires storing the previous combination to generate the next one. Therefore, the space required for storage is $2 \times k \times log(n)$ bits. But this is a sequential approach and dividing work among multiple available threads cannot be done efficiently using this technique.

C. Naïve division of combination testing among available threads

An naïve approach that would help divide the work among threads is to generate combinations based on the starting numbers, and split the work according to thread id's matching node numbers. There would be $n - k + 1$ such starting combinations, and hence the same number of threads are required. In such a case the number of threads is in the order of n. If there are more number of threads available, then combinations can be generated with slight modifications, say considering first 2 nodes different for each thread thereby utilizing n^2 threads. However, this approach leads to uneven distribution of work among threads, with threads having *id* numbers in the beginning doing more work than the ones with lager *id* numbers.

D. Equal work division among all available threads

The easiest way to ensure all threads do same amount of work is by calculating the total number of tests possible and then dividing the work among all available threads. By taking this approach all threads are now responsible for an equal amount of work (some threads might have to do a single test more if the number of threads does not divide the total work evenly). Now, the naive approach to solving the problem using the above technique is to generate all the possible combinations to test and store them in a data structure, and let the threads access the specific combinations for testing. However, this approach would require a lot of space to store all the combinations in the first place as discussed before.

Therefore, the ideal approach would be able to generate the combinations during runtime and perform the testing. Since, threads would require random lexicographical combinations to work on, generating the same efficiently is important. Generating a specific combination using the index in the lexicographic order can be done efficiently [3]. There exists a mapping from natural numbers i.e., indices in the lexicographic order to combinations, and this methodology is also known as *combinadics*.

Using combinadics, lexicographic ordering for any index can be found from a single set of elements (which can be node numbers in the case of graph problems.) However, while using BFS-tree information for testing reduced number of combinations, there are additional constraints with respect to the number of sets of elements and number of items chosen from each.

IX. Using Memory Coalescing

In certain cases, where the size of the data required for computation is larger than that of the shared memory, even when using the most efficient data structures, storing and accessing the data from the global memory is required. Due to the increased latency, as evident from the experimental results in [2], the time required to access data from the global memory is much larger than that from the shared memory. Therefore, to reduce the penalty in execution time while using the global memory, certain primitives like *Memory Coalescing*, available for the Nvidia GPUs can be used.

Given problem λ, let the data required for computation be stored in the global memory, and total number of operations be given by λ_t. The number of computations can be further divided into ω sets, given by the Equation [7]:

$$\lambda_t = \sum_{i=1}^{\omega} C_i \tag{7}$$

where, C_i is the number of computations in set ω_i. Let κ_{C_i} be the total number of global memory accesses required to complete the computations for C_i. Also, let κ be the total number of global memory accesses required to compute λ, and the value is given by the Equation [8]:

$$\kappa = \sum_{i=1}^{\omega} \delta_i \tag{8}$$

where, δ_i is the number of global memory accesses required to perform C_i. Since, global memory accesses are expensive, the objective here is to organize the data so that κ is minimized. Data from the global memory is accessed in the form of transactions. Therefore, minimizing the number of global memory accesses is equivalent to minimizing the number of transactions.

Fig. 4 shows an example of memory access by threads of an active half-warp. In this case, each of the threads access data from different segments, thereby requiring the maximum number of transactions to fetch the data for computation from the global memory.

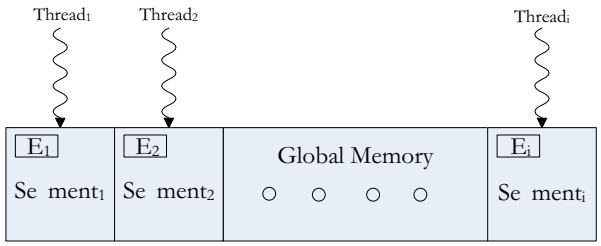

Fig. 4. Global memory access: Maximum transactions

The global memory access by 16 threads (half-warp) is coalesced into a single memory transaction if data accessed by all threads lie in the same segment. It must be noted that while data elements might be contiguous, they need not be in the

same segment. For example, there might be a single transaction required to access 64-bytes of data if it is stored entirely in a single segment. But, for another set of data, accessing a set of 50-bytes might require more transactions if the data is split over multiple segments in the global memory.

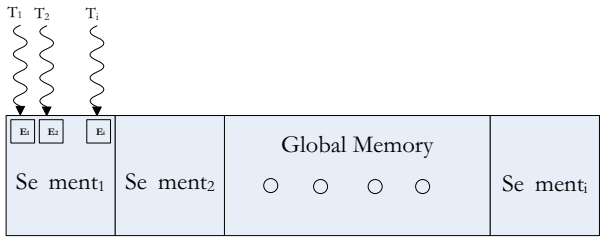

Fig. 5. Memory coalescing in effect: Minimum transactions

An example of memory access by threads resulting in memory coalescing is shown in Fig. 5. In this case, the data required by all the threads in the half-warp is fetched from the global memory using a single transaction.

The number of transactions required to access data from the global memory using memory coalescing depends on a number of factors like the compute capability of the system in question and whether the data that is being accessed by the different threads within the active warp is aligned and sequential or non-sequential [6]. A comparison for the different available options is shown in Table III.

TABLE III
MEMORY TRANSACTIONS AND COMPUTE CAPABILITY

Compute Capability	Access Pattern	Data Size in Bytes	Memory Transactions
1.0	Sequential	128	2
1.1	Sequential	128	2
1.2	Sequential	128	2
1.3	Sequential	128	2
2.0	Sequential	128	1
1.0	Non-sequential	128	32
1.1	Non-sequential	128	32
1.2	Non-sequential	128	2
1.3	Non-sequential	128	2
2.0	Non-sequential	128	1

From the data available in Table III, it is evident that for CUDA compute capability versions 1.2 and later, accessing non-sequential data is handled in the same manner as sequential data.

X. AVOIDING PARTITION CAMPING

Although the shared memory has low access latency, bank conflicts reduces the efficiency. Similarly, from the global memory perspective, memory coalescing is the performance booster while partition camping is the limiting factor. Memory coalescing and partition camping deal with data transfers between the global and on-chip memories, while bank conflicts deal with data access from on-chip shared memory. The shared memory on the Nvidia systems is divided into 16 (or 32) banks of 32-bit width. Similarly, the global memory is divided into 6 (or 8) partitions on 8- and 9-series GPUs (or 200- and 10-series GPUs) of 256-byte width.

The total time to access the data from the shared memory is indirectly proportional to the number of banks accessed by the threads in the active half-warp, and is given in the form of the following equation

$$\sum_{i=1}^{\beta} T_i \propto \frac{D_e}{\sum_{i=1}^{\beta} B_i} \tag{9}$$

where, T_i is the time required for thread i, β is the size of the half-warp, D_e is the number of data elements accessed by the threads, and $\sum_{i=1}^{\beta} B_i$ is the total number of *distinct* banks accessed by all the threads for computation on the data.

Partition camping occurs when global memory accesses are mapped into a subset of partitions, causing requests to queue up at some partitions while other partitions go unused. Therefore, it is analogous to shared memory bank conflicts, but on a wider scale. However, for devices of compute capability 2.x or higher, the effect of partition camping is taken care of by cached memory reads from the global memory. As shown in Equation [9] for shared memory bank conflicts, the total time to access the data from the global memory is indirectly proportional to the number of partitions accessed by the threads in all the active half-warps, and is given in the form of the following equation

$$\sum_{i=1}^{\gamma} T_{iw} \propto \frac{\sum_{i=1}^{\gamma} CM_i}{\sum_{i=1}^{\gamma} Part_i} \tag{10}$$

where, T_{iw} is the time required for threads in the active warp W_i, CM_i is the total number of coalesced memory access required to access the data elements to be processed by the threads in active warp W_i, and $\sum_{i=1}^{\gamma} Part_i$ is the total number of *distinct* partitions accessed by all the threads in warp W_i for computation on the data, and γ gives the total number of active warps. The objective of accessing the data from the global memory efficiently is to *Minimize*($\sum_{i=1}^{\gamma} T_{iw}$), which is equivalent to *Maximize*($\sum_{i=1}^{\gamma} Part_i$).

In Fig. 6, partition camping effect is illustrated with an example. Let the streaming multi-processors in the Nvidia system be denoted by SM_i, where $1 \leq i \leq 30$. The active warps are given by W_i, where i is the streaming multiprocessor on which it is being executed. Now, if the data in the global memory is distributed in such a manner, that for a given instance of execution, all the active warps access data from the same partition, in this case $Partition_1$, then partition camping takes place. The data access is sequentialized for each of the warps, and all the other partitions are not accessed. The active warps accessing a particular partition are shown in a table inside the partition.

In Fig. 7, a case where partition camping is avoided is shown. In this case, all the active warps are distributed evenly amongst the available partitions and the mapping is given by the following equation

$$Partition_{i\%p} \Leftarrow W_i \tag{11}$$

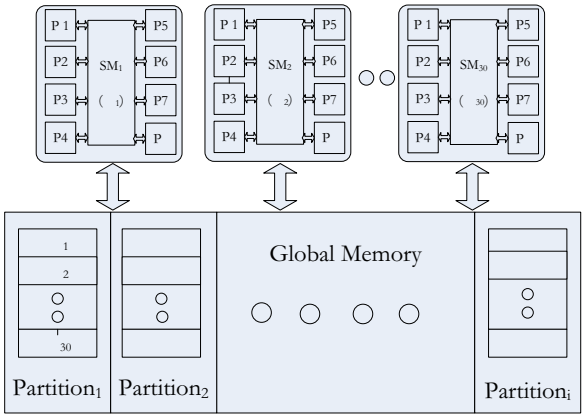

Fig. 6. Partition camping in effect

where, p is the total number of partitions available.

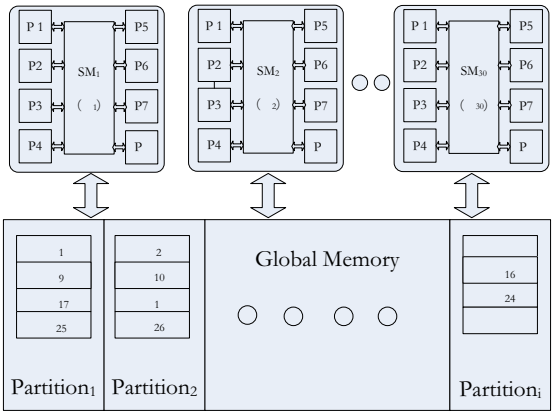

Fig. 7. Avoiding partition camping by distribution of warps

A. Avoiding Partition Camping While Accessing Data for Graph Problems

Partition camping takes place due to memory access patterns among different active warps. For solving the counting problem on triangles as given in Algorithm 2, different sets of levels are computed upon by different warps being executed on available streaming multiprocessors. Due to the pattern of the data being accessed, adjacent sets of levels have some shared levels (1 in the case of counting triangles, k-cliques, k-independent sets and k - 1 in the case of connected subgraphs of size k.) Therefore, during computation, due to the amount of shared data, threads from different active warps might need to access the same information leading to accessing the same partition in the global memory, thereby causing partition camping.

The analysis of the issue can be illustrated using the example of the BFS-tree given in Fig. 3. Now, for finding triangles, as given in the Algorithm 2, different warps would work on different adjacent level sets (ALS) at the same time. Let us assume, warp W_i operates on a set of levels given by ALS_i and is executed on the streaming multiprocessor SM_i. For the graph in Fig. 3, let us consider the second and the third sets of levels. When threads operating on these sets from different warps access the data related to the common nodes (numbered 4 to 8) simultaneously, partition camping occurs. The potential for such partition camping results from using a single adjacency matrix for the entire graph as the data structure, as shown in Fig. 8. It must be noted that the same problem would arise by using other data structures too, provided the entire data is stored together.

Keeping relevant data for the adjacent level sets separately in different partitions solves the above issue. For example, the data in the regions of intersection i.e., between data for ALS_1 and ALS_2, and also between ALS_2 and ALS_3 would have to be duplicated. Also, the design would be efficient if the data for a specific set of levels do not span across partitions. Therefore, the width of the sets of adjacency information data must be a maximum of 256-bytes. An arrangement of the data using the above mentioned idea is shown in Fig. 9.

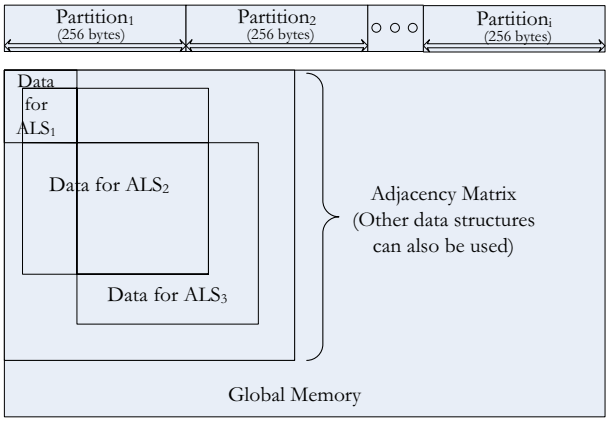

Fig. 8. Data structure with potential for partition camping

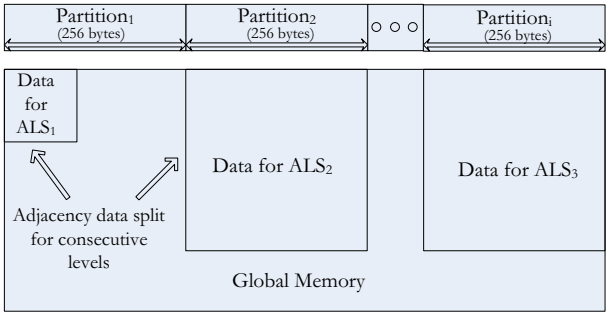

Fig. 9. Storing redundant information for avoiding partition camping

XI. Experimental results

The triangle counting problem as described in Algorithm 2 is implemented using both CPU and GPU. The CPU consists of quad-core 2.27 GHz Intel Xeon processors, with 12 GB of memory. The GPU used for the experiments is Nvidia C1060 card, with 4GB of memory. The CPU implementation is performed using a single thread.

Triangles are counted in graphs of sizes ranging from 200 to 1200 nodes. The timings for the CPU and GPU implementation are plotted and compared in Fig. 10. For implementing triangle counting using Algorithm 2, a BFS-tree for the input graph is required which is generated using Algorithm 1. Therefore, the timings for counting triangles include the executing time for both Algorithms 1 and 2.

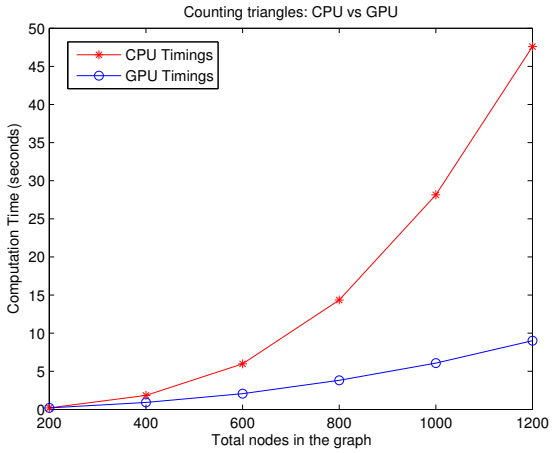

Fig. 10. Comparing timings for counting triangles using CPU and GPU

For smaller size graphs, due to overhead in transferring data from the host i.e., the CPU memory to the device i.e., the GPU memory, the timings are almost similar. But, as can be observed from the plots in Fig. 10, the performance of the GPU increases with the number of nodes in the graph. For 1000 nodes or more, there is 5–6 times improvement in the timings of the GPU as compared to the CPU.

Experiments were also performed using the data available on the Stanford Network Analysis Project [11]. Using reasonably larger graphs of size ranging from $5,000$ to $25,000$ nodes, it can be observed, as shown in Fig. 11, that the computation on the GPU is attains a 10 times speedup as compared to that on the CPU. For graphs of size $100,000$, the time required for the computation on the GPU is about 170–180 seconds.

The triangle counting algorithm is then implemented on the GPU using modified data structures to incorporate the usage of available primitives like memory access coalescing and avoiding partition camping, as discussed in the earlier Sections. The timings for the implementation using naïve data structures and modified ones with redundant data are plotted for comparison in Fig. 12. The experiments are done on input graphs of size ranging from 200 to 1200 nodes.

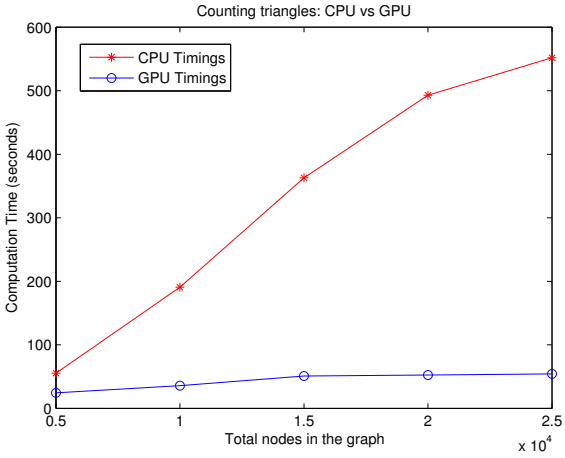

Fig. 11. Comparing timings for larger graphs

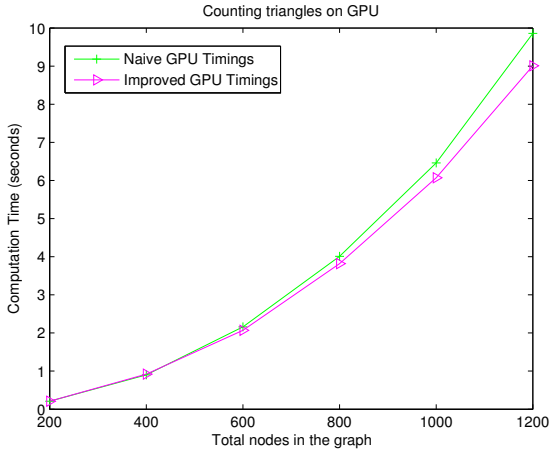

Fig. 12. Counting triangles using global memory with memory access coalescing and avoiding partition camping

As can be observed from the plots in Fig. 12, the performance of the GPU increases when making use of the available primitives and approximately 6–8 % performance gain is achieved over the naïve implementation. Therefore, the triangle counting problem achieves high speed-up from being solved on the GPU as compared to the CPU, and additionally the efficiency of the naïve implementation is further improved by using the available memory access primitives for the global memory effectively.

XII. Conclusion

In this paper, techniques to solve graph problems on large data sets using GPUs are discussed and analyzed. Data that cannot be stored in the shared memory while using the most efficient data structure, has to be stored in the global memory and fetched from there during computations. Therefore, accessing data efficiently from the global memory to offset the increased memory latency is essential. Hence, procedures to utilize the different available primitives for storing and retriev-

ing data efficiently from the global memory such as memory coalescing and avoiding partition camping are studied.

For studying the properties of graphs, generating combinations of nodes is required. The overhead of combination generation can be significant, both from the required time and space perspective. Therefore, techniques to efficiently generate combinations of nodes to be tested on graphs have been discussed in this paper.

An algorithm for triangle counting in a given graph utilizing all the above mentioned primitives is analyzed and implemented, and the results are reported. As evident from the experimental results, by properly utilizing the above methodologies, access time for retrieving data from the global memory is effectively reduced and improves the performance by a significant amount as compared to the naïve implementation. The triangle counting algorithm is also implemented on the CPU, and comparative analysis of the results are done with those of the GPU.

Our future work would involve handling streaming graphs that are much larger in size, and need to be stored externally on disks or tapes, and also designing heuristics to take advantage of existing primitives for solving problems belonging to other domains.

REFERENCES

[1] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM International conference on Knowledge discovery and data mining*, pages 16–24, 2008.

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

[3] B. P. Buckles and M. Lybanon. Algorithm 515: Generation of a Vector from the Lexicographical Index [G6]. *ACM Transactions on Mathematical Software*, 3(2):180–182, June 1977.

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

[5] 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.

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

[7] J. Grabowski and M. Wodecki. A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. In *Computers & Operations Research*, pages 1891–1909, 2004.

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

[9] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

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

[11] 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.

[12] Charles J. Mifsud. Algorithm 154: Combination in Lexicographical Order. *Communications of the ACM*, 6(3):103–105, March 1963.

[13] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in CUDA. *NVIDIA Technical Report*, 2009.

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

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

[16] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. DOULION: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 837–846, New York, NY, USA, 2009. ACM.

[17] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *High Performance Graphics'09*, pages 167–171, 2009.

[18] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[19] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending Against Sybil Attacks via Social Networks. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 267–278, New York, NY, USA, 2006. ACM.