International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.8, Issue.9, September 2020 E-ISSN: 2347-2693

Compressing Graphs Using Quadtrees for Efficient Computation on GPUS

Amlan Chatterjee

Dept. of Computer Science, California State University Dominguez Hills, Carson CA, USA

Author's Mail: achatterjee@csudh.edu, Tel: +001 310-243-3240

DOI: https://doi.org/10.26438/ijcse/v8i9.1118 | Available online at: www.ijcseonline.org

Received: 05/Sept/2020, Accepted: 15/Sept/2020, Published: 30/Sept/2020

***Abstract*—** Exploiting the computation potential of multi-core Graphics Processing Units (GPUs) requires reducing memory access latency and memory transfer overheads. Although the GPUs provide fast processing capabilities, the memory on such devices is significantly less. Though the memory hierarchy of GPUs provide certain fast levels, these are limited in terms of storage. Also, in order to use the GPUs for computation, the data must be transferred into the memory of the same; hence, in order to reduce the memory latency for large volume of data transfer, efficient techniques are required. Analysis of graph data on GPUs have many practical applications, and has been studied by both academia and industry. The graph data must be stored on the GPU memory to perform computation and analysis on the same. There are different data structures that can be used to store graphs in the GPU memory. Employing compression techniques to reduce the size required by the data is useful; however, the computation must be performed on the compressed data itself since decompressing the data would not be feasible. In addition to saving space on the device, using compressed data structures also reduces the memory transfer overheads both between the CPU & GPU, and between the various levels in the memory hierarchy of the GPU, thereby compensating for some of the additional time to retrieve information from the compressed data. Storing data using efficient compression techniques and operating on the compressed data is therefore useful. Quadtree data structures are generally used for storing and representing images for various applications. However, graphs when represented as adjacency matrix are comparable to images; hence, using recursive partitioning techniques, the data can be effectively compressed. In this paper, we show techniques based on quadtrees to efficiently compress graph data for storing and computation on GPUs. Additional techniques are also introduced which result in hybrid data structures that perform better for specific cases. Empirical results show 80-90% decrease in the space requirements to store graphs with real-world properties.

***Keywords*—**Compression, Quadtree, Graph Compression, GPUs

I. INTRODUCTION

Graphs represent data from a wide domain of applications. In general, the graph data is stored using either an adjacency matrix or an adjacency list. Based on the sparsity of the graph and special characteristics, the choice varies between the two. For sparse graphs, where there are significant number of 0 values, storing the data in an adjacency list is efficient. For denser graphs, where storing each element i.e., edge information separately would result in a lot more memory usage, using an adjacency matrix is effective. In any case, reducing the memory requirement overhead for computation is required and is a challenge that needs to be addressed using advanced techniques.

Solving graph problems on GPUs have many applications [1] [2] [3] [4]. Nvidia GPUs with Compute Unified Device Architecture (CUDA) can be employed for solving many general purpose real-world problems. For the purpose of computation using GPUs, the required data must be copied from the CPU and stored on the GPU memory; the objective is to store the data in the level of memory in the hierarchy which has the least memory access overhead latency [5]. Also, since copying data from the CPU to the GPU adds a significant overhead, being able to reduce the total data required to be transferred between the host and the device increases the performance gained by transferring the computation to the device. There are different techniques that can be employed to perform computation on large graphs [6] [7][8]. But even with dividing the graph data using vertical or horizontal splitting, the resulting components of data that is required for any computation might not fit in the memory level with least access latency, or in the worst case might not fit on the device at all. Therefore, efficient techniques need to be designed to compress the data and store it in the given memory.

The compressed data should fit on the memory; however, some techniques require decompressing the data before it can be used for computation. Such techniques would be limited in applications in the case of limited memory. Therefore, it is essential for computations to be performed on the compressed data rather than decompressing it. In

addition, there would be added complexity for retrieving adjacency information from compressed data, which needs to be taken into account when designing the algorithms for compression.

In this paper, we propose using quadtree based compression techniques to effectively reduce the storage requirements for graphs to be computed using GPUs. Following is the outline of the paper. Related work on graph compression is presented in Section II. This Section also discusses methods related to compression of data irrespective of the domain it belongs to. In Section III, we provide an algorithm to generate quadtree for a given graph from its adjacency matrix. In Section IV, we provide an algorithm that introduces a hybrid approach that modifies pure quadtree and combines it with other data structures to improve space requirements. Implementation results are discussed in Section V. Conclusion is presented in Section VI.

II. RELATED WORK

Graph compression techniques like other methods for compression can be broadly classified into either lossy or lossless compression. For lossy compression techniques, the data in the original form cannot be fully recovered. In case of lossless techniques, after decompression or when accessing the compressed data, the adjacency information in the original form can be fully recovered. For sensitive data, lossless techniques are favored; while lossy techniques can be used for storing or transmitting data that has a higher threshold for errors. In this paper, we focus our techniques on graphs that store data which are sensitive and therefore required lossless compression.

A. Classification of techniques

In general, the majority of compression algorithms for graph data can be broadly classified into the following groups.

• Identifying and Replacing common structures: Graphs can have certain structures in them which can be easily identified and replaced with something that takes a smaller memory. For example, if a graph contains a clique, which is a completely connected subgraph, then instead of storing the adjacency information with all the nodes and edges, an identifier in the form of a special node can be stored to represent the specific structure.
• Similar Adjacency Information: Graphs can contain nodes that share similar adjacency information with certain other nodes. Instead of storing the same information for a set of nodes, an identifier can be stored instead which indicates the presence of certain common adjacency information. Even differential techniques can be employed, where minor differences can be encoded instead of storing the entire information for neighborhoods that differ in the adjacency information within certain predefined threshold.
Based on the above mentioned principles, various techniques have been proposed and studied. Some techniques work well for graphs with specific properties. However, the initial overhead of processing the graph data to aide in the compression should also be taken into account while calculating the overall effectiveness of any method.

B. Graph Compression Related work

Techniques to address graph compression has been developed over long time and there are many that have been studied. Finding and replacing common structures with an identifier that requires less space is one such method. In a complete bipartite graph $K_{m,n}$ the m×n edges need not be stored separately and can be replaced using a special node and additional m+n edges [9] [10].

A graph can be used to model the web, where the nodes are the different addresses and the edges are the links between the same. In this case, there are many subgraphs that share the same adjacency information. Hence, using pointers to similar adjacency information and differential techniques for the additions and deletions help gain significant compression [11].

Exploiting locality information and using common neighborhoods can be an effective technique that specifically works well for power law graphs and can be used to compress the same [12].

For graphs that exhibit a natural order, leveraging lexicographic ordering along with adjacency information is an efficient technique in compressing graphs [13]. Using separators, i.e., a subset of vertices whose removal does not disconnect the graph, is also an efficient technique [14]. Even for separators that disconnect the graph, using a recursive framework on the resulting components can achieve compression of the graph data.

Graph compression techniques are also employed to study biological networks. Since it is difficult to compare large biological networks with one another, using data compression to reduce the size of the data helps in comparative study [15].

In our previous work we discuss solving graph problems on GPUs. We have earlier introduced a number of algorithms and data structures for efficient computation on graph data [16]. In this paper, we extend our work to increase the amount of data that can be stored on the GPUs for computation by introducing quadtree based graph compression.

III. QUADTREE REPRESENTATION

Quadtrees are generally used to represent images in two dimensional space. The basic idea is to limit the number of significant values in any region. For example, in the most common form of quadtree, which is a point-region quadtree of in short PR quadtree, the target is to limit the number of significant values to 1 in any region. This is

done by recursively sub-dividing the regions into quadrants, until there is an acceptable number of significant values in each. Once a region or quadrant reaches the target, further sub-division is not required; for other regions, the process continues. The regions are represented as nodes in the quadtree, and each internal node has exactly 4 children [17].

Specifically, for the PR-quadtree, the target number of significant values in a region is limited to 1. Therefore, the regions are recursively sub-divided until the number of points in each region is 1 or less. The equivalent quadtree for the sample points shown in Fig. 1 is given in Fig. 2.

A. Graphs as Quadtree

Graphs can be stored using Quadtree representation. Con sidering the adjacency matrix, the neighborhood information is stored as 0's and 1's. Now, using certain identifiers, the adjacency matrix can be represented as quadrants of data. A quadrant can be represented as a leaf in the quadtree representation if it matches with the data corresponding to the identifier.

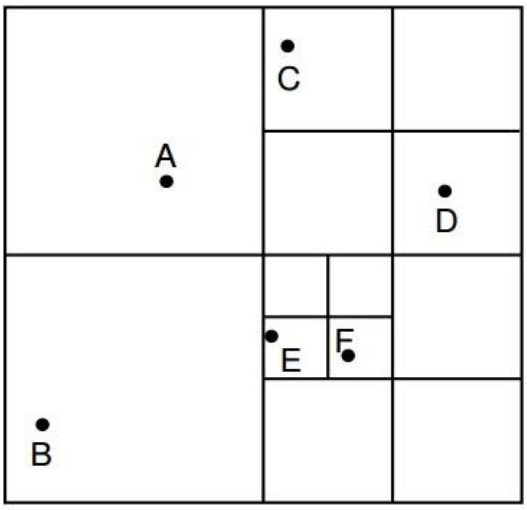

Fig. 1. Sample data points for a 2-D region

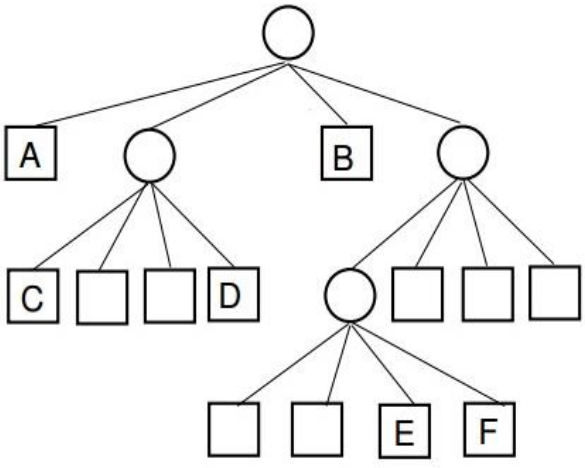

Fig. 2. The PR-Quadtree for the region shown in Fig. 1

In other cases, further sub-divisions are required. Hence, using such a method, the adjacency information of the graph is represented as a Quadtree which is a bit array. The bit array consists of the following data for each of the quadrants:

- 0: all 0's
- 1: all 1's
- 2: 0's in diagonal, and rest 1's
- 3: sub-division required

Because this technique considers only 4 types of values, 2 bits can be used to represent each quadrant in the bit array. As an example, consider the graph shown in Fig. 3.

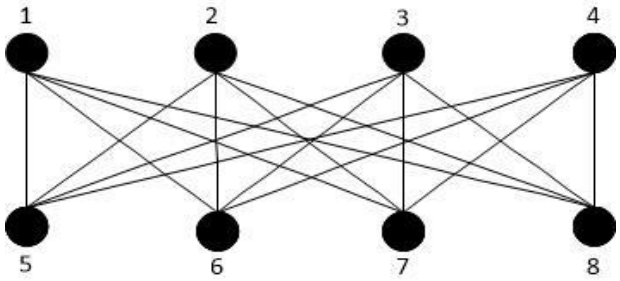

Fig. 3. A sample graph

The adjacency matrix of the graph is taken into consideration for converting the graph data into an equivalent quadtree. The adjacency matrix for the graph in Fig. 3 is given in Fig. 4.

0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0

Fig. 4. Adjacency matrix of graph shown in Fig. 3

The quadtree representation of the given graph is shown as Follows in Fig. 5.

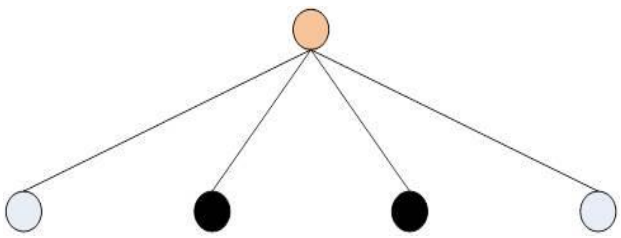

Fig. 5. Quadtree representation of graph shown in Fig. 3

The quadtree contains 4 nodes in the representation. The first node indicates there is no match, so a corresponding value of 3 is added to the bit array and further sub-division is done. For the resultant quadrants, each matches with one of the identifiers. Finally, the quadtree byte stream is given by Q = {3,0,1,1,0}. The values of 0, 1, 1 and 0 represents the data after the division and corresponds to respectively the top-left, top-right, bottom-left and bottom-right quadrants.

The method for quadtree generation is given in Algorithm 1. The input is the adjacency matrix representation of the graph. The method *CheckUniformity* checks whether the given adjacency matrix data matches any of the pre-defined uniform data values i.e., all 0's or all 1's etc. Based on the provided data, the checking is done and the identifier corresponding to the matching pattern is returned. For example, a 0 is returned for all 0's, a 1 is returned for all 1's, a 2 is returned for diagonal 0 values and remaining 1's and a 3 is returned if no match is found. If a value of 0, 1 or 2 is returned from the method,

Algorithm 1: QuadGen: Quadtree generation from adjacency matrix

```
Input: Adjacency matrix A[ ][ ] of graph G
Output: Quadtree Q of G
begin
    Result ← CheckUniformity(A);
    if Result = 0 then
        Add(Q, 0);
    else if Result = 1 then
        Add(Q, 1);
    else if Result = 2 then
        Add(Q, 2);
    else
        Quads{} ← DivideIntoQuadrants(A) ;
        forall Quadrants_i ∈ Quads{} do
            QuadGen (Quadrants_i);
    Output ← Q;
```

then it is added to the bit representation of the quadtree; for a returned value of 3, the Algorithm 1 is called recursively for each of the resulting quadrants of the matrix that are generated by using the method *DivideIntoQuadrants*. In this case, the algorithm loops over total of n 2 elements in each of the levels of the quadtree, where $|V| = n$ for the given graph $G = (V,E)$. For the worst case scenario, there are $\log_2 n$ levels in the quadtree; hence for the algorithm to traverse through the levels and generate the quadtree, the time complexity is given by $O(n^2 \log_2 n)$.

B. Compression leveraging quadtree

Consider a sample graph of 8 nodes as shown in Fig. 3. The adjacency matrix information for the corresponding graph is given in Fig. 4. Assuming each of the values in the adjacency matrix can be stored in a single bit, the number of bits required to store the adjacency matrix information is 64 bits. Now, considering the quadtree representation of the same graph, it can be seen in Fig. 5, there are only 5 elements in the tree. Taking into account our representation, where each value takes 2 bits, the total number of bits required is 10. Therefore, from this example it is evident that efficient compression can be achieved by leveraging the quadtree representation of graphs.

C. Numbering matters

In this sub-section we discuss how even with the same structure of the graph, the numbering of the nodes can play a significant role in the achieved compression using quadtrees. Considering the graph shown in Fig. 6 and the corresponding adjacency matrix and quadtree representation. The structure of the graph is same as the one considered before as shown in Fig. 3.

However, due to the difference in numbering and the corresponding change in the adjacency matrix structure, this quadtree has 21 elements instead of the 5 before, and therefore requires 42 bits instead of the 10. In the case of the renumbered graph,

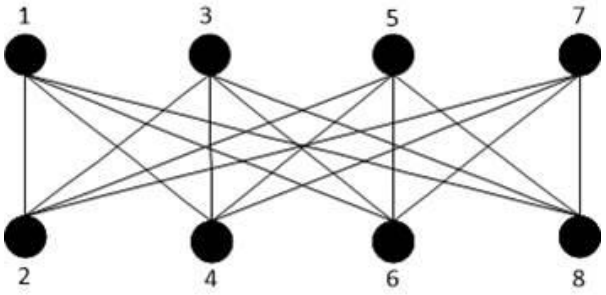

Fig. 6. Sample graph with nodes numbered in a specific way

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

Fig. 7. Adjacency matrix for the sample graph shown in Fig. 6

since the adjacency matrix changed, the corresponding quadtree also changed resulting in having 21 elements compared to just 5 elements. Now, for the quadtree representation, the size of the resultant structure is directly proportional to the number of matching quadrants. Since the structure of the adjacency matrix is dependent on the

node numbering, even with the same structure, the adjacency matrix for some cases might have a better match with the patterns as compared to others. Therefore, the numbering of the nodes in the graphs have a significant effect in the overall compression achieved by using the quadtree representation.

IV. HYBRID APPROACH

The quadtree representation requires uniformity in the quadrants or specifically in the case of PR-Quadtrees, a single point in a region for the recursive sub-division to stop. However, it can be noted that when recursive sub-division occurs for quadrants containing smaller number of bits, the memory required to store the quadtree representation is more than what is needed to store the raw data itself. Hence, the quadtree representation is effective when large quadrants of data can be replaced using identifiers; however, is inefficient for smaller sized quadrants that require sub-division. So, we propose a hybrid approach, where the larger quadrants are represented as nodes in the quadtree; but once the size of the quadrants reach a pre-defined threshold, instead of further sub-division, the raw data is stored. Although this adds an overhead identifier required to differentiate between the quadtree data representation and raw bits, the overall memory requirements are decreased. The result of this hybrid approach is verified by the performed experiments and reported in the results section.

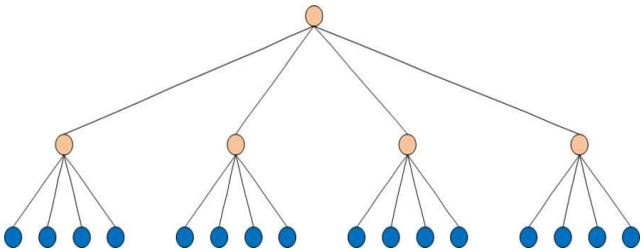

Fig. 8. Quadtree representation for the sample graph shown in Fig. 6

V. EXPERIMENTAL RESULTS

We study the characteristics of real-world graphs to identify properties that might be suitable for compression using quadtrees. We refer to real world data sets for the graphs to be considered [18]. The Stanford Network Analysis Project (SNAP) contains the Stanford Large Network Dataset Collection framework, which provides a large collection of real- world graphs [19]. Out of the various data sets available, to study the properties of the graphs, for this paper we consider the following three types of networks:

1. Road Networks: We consider 3 different road networks from the dataset. The Texas Road Network (TRN), Pennsylvania Road Network (PSN) and California Road Network (CRN). Each of these graphs represent the road networks of the respective states. The actual data includes information about the road intersections and connections. For the graph, the intersections are represented as nodes, and the connecting roads as edges. All the edges are assumed to be undirected.

2. Email Network: To represent interactions and communication, the Enron Email Network (EEN) is considered. The graph consists of the email communication between the employees of Enron; each employee email address is a node in the graph, and information exchange is represented by the edges. The size of the dataset that consists of the communication emails is about half million. The data is also represented as an undirected graph.

3. Social Circles: To represent social interactions, the Facebook Social Circles (FSC) graph is considered. The data consists of interactions between people on the online social network platform, and the circles represent the group of people or "Friends" in the context. This graph is undirected and the data is gathered using a specific application from survey participants.

The data for the different types of graphs considered in this paper are shown in Table I. The total number of nodes, number of edges and the number of nodes in the largest connected component for each of the networks are presented in the table.

TABLE I
REAL WORLD GRAPHS

Graph	Nodes	Edges	Nodes in largest connected component
TRN	1,379,917	3,843,320	1,351,137
PRN	1,088,092	3,083,796	1,087,562
CRN	1,965,206	5,533,214	1,957,027
EEN	36,692	367,662	33,696
FSC	4,039	176,468	4,039

In our earlier work, we have used breadth-first search (BFS) based techniques to perform analysis on real-world data [16]. We have shown earlier, that BFS tree properties can be leveraged to reduce the space requirements. For solving problems on graphs, on a portion of the data that is of interest is required for computation. Therefore, the entire graph can be divided using BFS tree levels, and the space required for storing each of the sets of levels is what needs to be focused on. However, further investigation reveals that even within the sets of data in the BFS tree, there might be more than one single connected component. For most graph problems being solved, having connections between nodes is a pre-requisite. Therefore, if there are disjoint sets of nodes within the graph as indicated by the number of connected components that do not share any connections with each other, then the computations can leverage this data. Since nodes belonging to different connected components are not part of most calculations, the adjacency data for such connected components are not required on the memory together. Using the BFS tree data, even further

analysis can be done on the number of connected components with 2 or 3 levels of the BFS tree [16]. The size of the largest components with the 2 or 3 levels of the BFS tree provides the memory required for computation on various graph problems, and is given in the Table II for the graphs being considered in this paper.

TABLE II
CONNECTED COMPONENTS IN BFS TREE LEVELS OF REAL WORLD GRAPHS

Graph	# of conn. comp. 2 levels	Nodes in largest connected component 2 levels	# of conn. comp. 3 levels	Nodes in largest connected component 3 levels
TRN	2,033	73	1,558	247
PRN	1,887	136	1,446	558
CRN	3,147	92	2,545	195
EEN	4,455	24,512	4,454	26,610
FSC	13	2,900	13	3,419

As discussed earlier, the characteristics of graphs are dependent on many factors, with the density being one of the most significant. Therefore, it is imperative to consider the density of the graph while deciding on the appropriate data structure to store and represent the same. Hence, analysis of the graph storage requirements for varying densities is required and provides valuable insights.

Different sizes of graphs in the powers of 2 are considered for the experimental analysis. The number of nodes are varied from 1024 to 8192, with the number being doubled for each separate case. These sizes of 1024, 2048, 4096 and 8192 are directly related to the actual graphs being considered. As evident from the discussion regarding size of connected components, these sizes relate directly to the largest connected component size of the graphs under consideration for this paper. For the given graphs, the density of the same is given by e/p ×100, where e denotes the number of edges, and p is the number of edges possible in the graph. In the experimental analysis, the densities are varied and the following percentages are considered 5, 10, 25, 50, 65 and 80.

Fig. 9 plots the sizes of the graph representations for adjacency matrix, quadtree (pure quadtree or PQT) and hybrid quadtree (HQT). The number of nodes are varied along with the density. It is evident from the graphs that the HQT requires less memory than the adjacency matrix or pure quadtree for all the densities and sizes, except for the 50% density; at this specific density the adjacency matrix representation is better. However, for smaller and larger densities, say below 25% and above 75%, the HQT outperforms the other representations. The outputs are as anticipated. Since the size depends on the uniformity of the quadrants, in case of graphs with low and high density, the adjacency matrix structure has more uniform quadrants or those that match the patterns then those graphs that have a density near 50%.

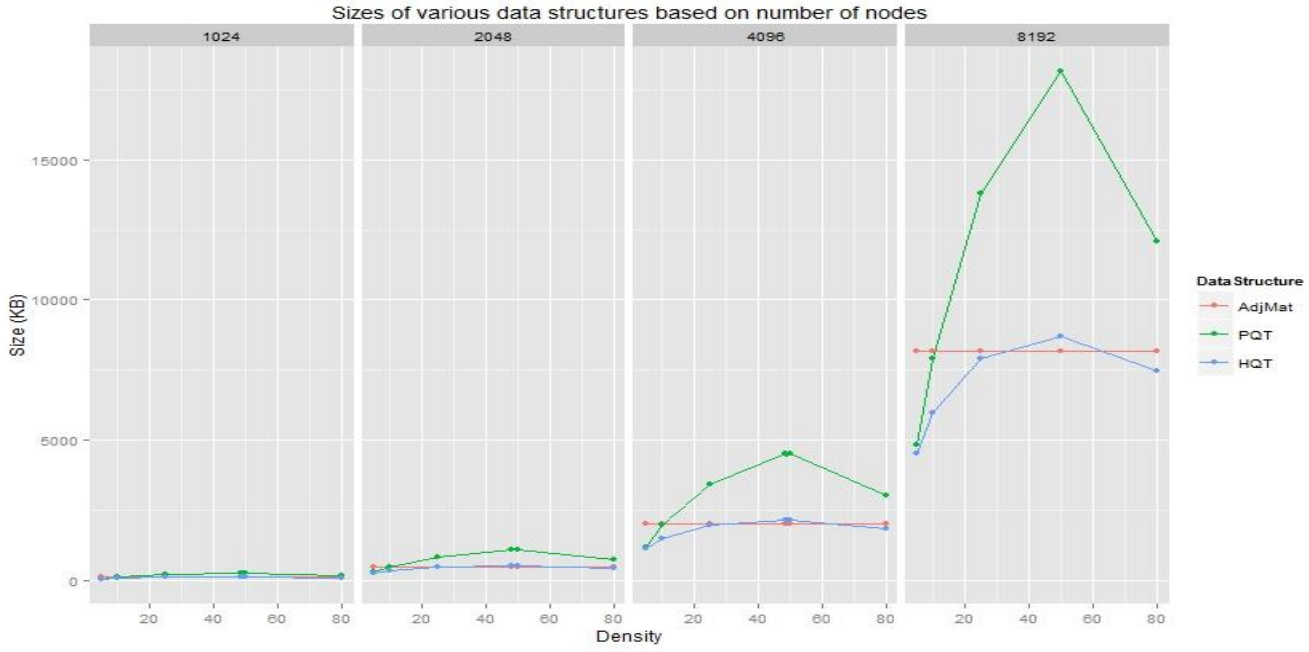

Fig. 9. Data representation comparison for high densities

In case of most real-world graphs, the data is sparse which is evident from the number of possible edges and the actual number of edges in the social networks or road networks under consideration [20] [21]. Hence, the proposed hybrid quadtree technique is also verified in cases of such data, where the density of the graph is low. For experimental purposes, the density of the graphs is varied from 0.10 to 1.20. The resulting quadtree sizes are plotted in Fig. 10, and the data shows that the quadtree representation actually performs better than the HQT or the adjacency matrix. The

HQT size requirements are similar to the pure quadtree for smaller size graphs of 1024 or 2048 nodes; however, for larger graphs of size 2048 or 8192 nodes, it performs worse than the pure quadtree. This can be attributed to the fact that the HQT requires the use of an extra identifier to indicate quadtree data or raw data. For all the graph sizes and low densities, the adjacency matrix performs worse as compared to the quadtree or HQT. In graphs of size 8192 nodes, for a density value of 1.20, the space required is almost 80% less than as compared to the adjacency matrix; for similar size graph, when the density is 0.10, the reduction in space requirement is 97%. Therefore, from the experimental results it can be concluded that using quadtree based compression techniques reduces the space required to store graph data by a significant amount. Hence, using such methods, storing and transferring graph data on the GPU memory and between the CPU and the GPU respectively, is highly efficient.

VI. CONCLUSION

In this paper we study graph compression techniques for efficient storage and computation using GPUs. We introduce quadtree based compression techniques and modifications of the same. A hybrid data structure composed of partly quadtree and partly adjacency matrix is also proposed. We study various special graphs, and analyze and identify methods for effective compression. Since the adjacency matrix and quadrant data dictates the structure of the quadtree and in essence the compression efficiency, it is also shown how numbering of the nodes in graphs influence the space required for the same. We focus on real-world graph properties by using breadth-first search tree based analysis results on the same. Different size graphs are chosen over varying densities and sizes, and are compared by performing empirical analysis on the same.

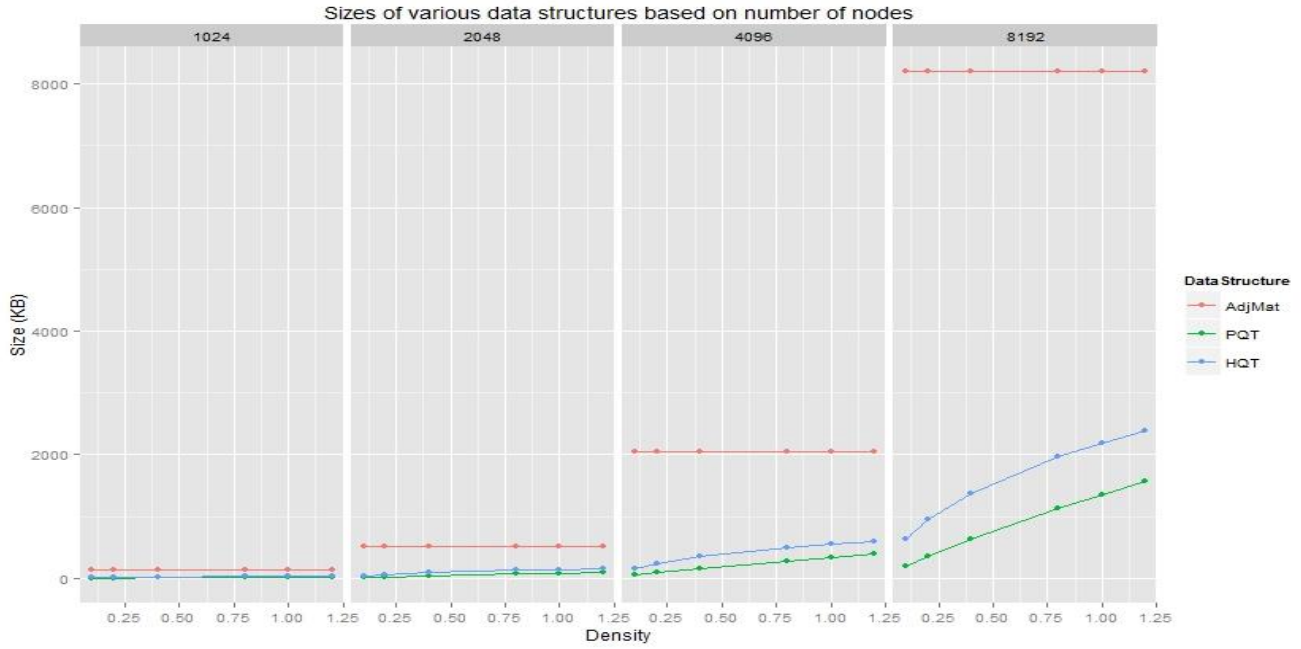

Fig. 10. Data representation comparison for low densities

From the results it can be concluded that the proposed techniques using quadtree is indeed an effective compression technique for storing graphs specifically on memory constrained devices.

As part of future work, we would like to study compression techniques on streaming graphs for performing analysis using GPUs. Also, applying the proposed compression methods to data belonging to other domains like biological networks can be explored.

REFERENCES

[1] A. Chatterjee, S. Radhakrishnan, and John K. Antonio. Counting Problems on Graphs: GPU Storage and Parallel Computing Techniques. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, pages **804**–**812**. IEEE, **2012**.

[2] 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**.

[3] 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.

[4] Amin Rezaeipanah, Mousa Mojarad, Link Prediction in Social Networks Using the Extraction of Graph Topological Features, International Journal of Scientific Research in Network Security and Communication, Vol.**7**, Issue.**5**, pp.**1-7, 2019**

[5] A. Chatterjee, S. Radhakrishnan, and J. K. Antonio. Data Structures and Algorithms for Counting Problems on Graphs using GPU. International Journal of Networking and Computing (IJNC), Vol. 3(2):pages **264**–**288**, **2013**.

[6] A. Chatterjee, S. Radhakrishnan, and J. K. Antonio. On Analyzing Large Graphs Using GPUs. In Parallel and

Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, pages 751–760. IEEE, **2013**.

[7] 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 SIGKDD international conference on Knowledge discovery and data mining, KDD '08, pages 16–24, New York, NY, USA, 2008. ACM.

[8] K. Parimala, G. Rajkumar, A. Ruba, S. Vijayalakshmi, Challenges and Opportunities with Big Data, International Journal of Scientific Research in Computer Science and Engineering, Vol.**5**, Issue.**5**, pp.**16**-**20**, **2017**

[9] Tomás Feder and Rajeev Motwani. Clique Partitions, Graph Compression and Speeding-up Algorithms. In Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing, STOC '91, pages 123–133, New York, NY, USA, 1991. ACM.

[10] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. In Proceedings of the 2008 International Conference on Web Search and Data Mining, WSDM '08, pages 95–106, New York, NY, USA, 2008. ACM.

[11] Keith H. Randall, Raymie Stata, Janet L. Wiener, and Rajiv G. Wickremesinghe. The link database: Fast access to graphs of the web. In Proceedings of the Data Compression Conference, DCC '02, pages 122– , Washington, DC, USA, 2002. IEEE Computer Society.

[12] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In Proceedings of the 13th International Conference on World Wide Web, WWW '04, pages 595–602, New York, NY, USA, 2004. ACM.

[13] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '09, pages 219–228, New York, NY, USA, 2009. ACM.

[14] Narsingh Deo and Bruce Litow. A structural approach to graph compression. In In MFCS Workshop on Communications, pages 91– 1**0**1, **1998**.

[15] Morihiro Hayashida and Tatsuya Akutsu. Comparing biological networks via graph compression. BMC Systems Biology, 4(Suppl 2), **2010**.

[16] A. Chatterjee, S. Radhakrishnan, and C. N. Sekharan. Connecting the dots: Triangle completion and related problems on large data sets using GPUs. In 2014 IEEE International Big Data Workshop on High Performance Big Graph Data Management, Analysis, and Mining, pages **1–8**. IEEE, **2014**.

[17] Hanan Samet. Using quadtrees to represent spatial data. In Herbert Freeman and GoffredoG. Pieroni, editors, Computer Architectures for Spatially Distributed Data, volume 18 of NATO ASI Series, pages **229**– **247**. Springer Berlin Heidelberg, **1985**.

[18] 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, **Vol. 6(1):29–123, 2009**.

[19] Leskovec, Jure, and Rok Sosič. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. ACM Transactions on Intelligent Systems and Technology (TIST) 8.1 **(2016): 1-20**.

[20] Deepayan Chakrabarti and Christos Faloutsos. Graph Mining: Laws, Generators, and Algorithms. ACM Computing Surveys, 38(1), June **2006**.

[21] J.M.Kleinberg, R.Kumar, P.Raghavan, S.Rajagopalan, and A.S.Tomkins. The web as a graph: measurements, models, and methods. In Proceedings of the 5th annual international conference on Computing and combinatorics, COCOON'99, pages 1–17, Berlin, Heidelberg, 1999. Springer-Verlag.

AUTHORS PROFILE

Dr. Amlan Chatterjee received his Masters in Computer Science from State University of New York at Buffalo, USA in 2009, and his Ph.D. in Computer Science from the University of Oklahoma, USA in 2014. Dr. Chatterjee is currently an Assistant Professor in the Department of Computer Science at California State University Dominguez Hills, USA. His research interests are in the areas of Big Data, High Performance Computing and IoT; his recent research focuses on graph compression and using GPUs to solve general-purpose problems. Dr. Chatterjee is a member of IEEE and ACM.

