Towards Efficient Graph Traversal using a Multi-GPU Cluster

Hina Hameed
Systems Research Lab, Department of Computer Science,
FAST National University of Computer and Emerging Sciences
Karachi, Pakistan

Sehrish Hina
Systems Research Lab, Department of Computer Science,
FAST National University of Computer and Emerging Sciences
Karachi, Pakistan

Nouman M Durrani
Systems Research Lab, Department of Computer Science,
FAST National University of Computer and Emerging Sciences
Karachi, Pakistan

Jawwad A. Shamsi
Systems Research Lab, Department of Computer Science,
FAST National University of Computer and Emerging Sciences
Karachi, Pakistan

***Abstract*—Graph processing has always been a challenge, as there are inherent complexities in it. These include scalability to larger data sets and clusters, dependencies between vertices in the graph, irregular memory accesses during processing and traversals, minimal locality of reference, etc. In literature, there are several implementations for parallel graph processing on single GPU systems but only few for single and multi-node multi-GPU systems. In this paper, the prospects of improvement in large graph traversals by utilizing multi-GPU cluster for Breadth First Search algorithm has been studied. In this regard, a DiGPU, a CUDA-based implementation for graph traversal in shared memory multi-GPU and distributed memory multi-GPU systems has been proposed. In this work, an open source software module has also been developed and verified through set of experiments. Further, evaluations have been demonstrated on local cluster as well as on CDER cluster. Finally, experimental analysis has been performed on several graph data sets using different system configurations to study the impact of load distribution with respect to GPU specification on performance of our implementation.**

***Keywords*—Graph processing; GPU cluster; distributed graph traversal API; CUDA; BFS; MPI**

I. INTRODUCTION

Data processing accompanied with GPGPU techniques is being employed to process large amount of data with limited resources in several application domains throughout the globe. However, there are several challenges when it comes to graph processing. These include dependencies between vertices in the graph, irregular memory accesses during graph processing, and scalability to larger data sets.

As graph problems grow larger in scale and more ambitious in their complexity, they easily outgrow the computation and memory capacities of single processors [1]. Given the success of parallel computing in many areas of scientific computing, parallel processing appears to be necessary to overcome the resource limitations of single processors in graph computations.

Utilization of parallel architectures became a viable mean in order to improve graph processing performance. However, besides the inherent complexities in graph traversal, parallelism is challenging in several aspects such as: finding the correct step to introduce parallelism in the algorithm, expensive memory accesses, communication overheads, poor locality of reference, and complex load balancing, etc.

Utilization of the distributed GPU clusters will make mining of large graphs faster and cheaper. Many Big Data applications such as Social networks analysis, traffic management, and disaster management systems that rely on graph traversals might be able to perform better, faster, and cheaper. In this context, Breadth First Search (BFS) and Single Source Shortest Path (SSSP) algorithms are important graph traversal algorithms which find their applications in several application domains such as the all pairs' shortest path algorithm, s-t shortest path algorithm, etc.

Since, motivated from the usability of BFS and potential of distributed graph processing, in this paper, a DiGPU - an API providing efficient implementations of these algorithms on distributed GPU clusters has been proposed.

DiGPU is a flexible user friendly CUDA-based implementation of BFS, which can be executed both on single node as well as multi-node systems. DiGPU can also be run on single node multi-GPU systems. For this purpose, it incorporates Unified Virtual Access (UVA) between host and device and Peer-to-Peer direct access between devices. In this work, the BFS algorithm has been implemented on single-node single GPU system as well as on single node multi-GPU systems. Further, the multi-node implementation of DiGPU has also planned that will help in efficient computation of large graphs.

This research has been supported by NVIDIA Teaching and Research Center Awards.

In this regard, the initial implementation has been tested on two types of experimental setups. The traditional hardware cluster comprises of combinations of NVIDIA Tesla K40 and NVIDIA GTX-780 GPUs. A test system on the GPU nodes of Georgia State University cluster has also been setup for further testing.

In summary, following are the main contributions of this paper:

1) The design and development of DiGPU has been proposed. It is an open source software module for distributed graph computations on a heterogeneous GPUs cluster.

2) The functions provided by DiGPU would become building blocks for many applications performing intensive graph computations or analysis. Availability of these building blocks will facilitate a user with the ease of designing graph applications.

3) A hybrid model has been developed. The model utilizes UVA and Peer-to-Peer access between GPUs on same node and CUDA-aware-MPI on different nodes of cluster.

4) Different experiments have been performed to study the impact of load-distribution according to specifications of GPUs on the cluster.

DiGPU will be beneficial for the community in computing large graphs over a series of GPU clusters. Our initial results are encouraging. In this work, it has been aimed to provide an open source version of DiGPU, to perform graph analysis which would make our work a true example of reusable graph building blocks for graph research community. The performance has been evaluated through an extensive evaluation over a large dataset and involving varying GPUs with heterogeneous computational power.

The rest of the paper is organized as: Section II covers the background and related work in this area; Section III illuminates our methodology and approach toward the research problem; Section IV contains details about our initial experimental setup; Section V lists down the results of our preliminary experiments and Section VI concludes this paper.

II. BACKGROUND AND RELATED WORK

This section extensively covers the related work that has previously been done regarding GPU implementations of SSSP and BFS algorithms.

A. Breadth First Search

The classic queue-based parallel BFS initiates the computation with a root node, putting that node in an empty queue. During each iteration, the head of queue is pulled out and all of the connected nodes are visited. Neighbor nodes visited for the first time are placed at the end of the queue. The output of the algorithm is an array storing the distance from the source or predecessor, for each vertex. The best time complexity reported for sequential algorithm is O(V+E).

The queue is a current level set of vertices. For each vertex in the current level, all its neighbors must be visited. The set of all neighbors composes the Next Level Frontier Set (NLFS). From the NLFS only new vertices are selected to build the queue for the next level. The BFS visit is divided into levels with a distance from the root that increases at each subsequent level [2].

Vibhav, *et al*. [2] performed BFS implementation for vertex compaction process. At particular time, small number of vertices may be active. They used prefix sum for assigning threads to active vertices only. They carried out experiments on various types of graphs and compared the results with the best sequential implementation of BFS and experiment shows lower performance on low degree graphs.

P. Harish, *et al*. [3] proposed accelerated large graph algorithm using CUDA. The proposed algorithms is capable of handling large graphs. In their implementation, one thread is assigned to every vertex. They have used frontier array, visited array and cost array which stores the minimum count of edges of every vertex from the source vertex S. During each iteration, every vertex checks frontier array index for itself and in case of positive values updates the cost of itself and its neighbors. But this algorithm's performance slacks due to large degree at few vertices. Also, since it loops the kernel which causes the more lookups to device memory, hence slowing down the kernel execution time.

Lijuan, Luo [4] proposed effective GPU implementation of BFS for designing an efficient queue structure. A hierarchical kernel arrangement is used in order to reduce synchronization overhead. Their experimentation results present similar computational complexity as the fastest CPU version with a potential speedup of up to and they claim to 10 times.

S. Hong, *et al*. implemented a novel warp-centric [5] method for reducing the inefficiency. Many graph algorithms suffer severe performance degradation in case of highly irregular graphs, i.e. when the distribution of degrees (number of edges per node) is highly skewed. Instead of assigning a different task to each thread, their approach allocates a chunk of tasks to each warp and executes distinct tasks in serial. They have utilized multiple threads in a warp for explicit SIMD operations only, thereby preventing branch-divergence altogether.

A. Grimshaw, *et al*. report [6] deals with parallel BFS on GPU clusters. Their work resorts to a duplicate removal procedure by using a heuristic that removes a high percentage of duplicates at the CTA level. In contrast, our algorithm also eliminates every duplicate in the Next Level Frontier Set (at a global level). They have used four GPUs that have a unified memory address space with a reduced latency, compared with a standard network interconnection.

Level Synchronous BFS [7] ensures the correctness of the computation by synchronization at the end of each level in a parallel implementation, . The number of levels is of the same order of the diameter of the graph, in real-world graphs, the computation is dominated by only two or three levels, for which the next level set of vertices is very large.

D. Merril, *et al*. [8] suggested that work-efficient parallel BFS algorithm should perform O(n+m) work. In order to achieve O(n+m) complexity, each iteration should examine only the edges and vertices in that iteration's logical edge and vertex-frontiers, respectively. For each iteration, tasks are

mapped to unexplored vertices in the input vertex-frontier queue. Their neighbors are inspected and the unvisited ones are placed into the output vertex-frontier queue for the next iteration. The typical approach for improving utilization is to reduce the task granularity to a homogenous size and then evenly distribute these smaller tasks among threads, by expanding and inspecting neighbors in parallel.

Leiserson and Schardl [2] designed an implementation for multi-socket systems that incorporates a novel multi-set data structure for tracking the vertex-frontier. In other implementations, hardware's full-empty bits are used for efficient queuing into an out-of core vertex frontier. Both approaches perform parallel adjacency list expansion, relying on runtimes to throttle edge-processing tasks in-core. Luo, *et al*. [4] present an implementation for GPUs that relies upon a hierarchical scheme for producing an out-of-core vertex frontier.

Talking about frameworks and APIs multi-GPU CUDA implementations of BFS have been provided in Medusa [9] and GunRock [10]. Both are graph processing frameworks capable of performing on multiple-GPUS on a single node. Medusa uses two schemes for multi-GPU execution: Replication – division of the graph into equal-sized partitions and store each partition on one GPU and maintain replicas of the head vertices of all cross-partition edges in the partitions where the tail vertices reside. Each cross partition edge is replicated in its tail partition, so messages are emitted directly from the replicas and every edge can access its head and tail vertices directly. The execution of is performed on each partition independently and the replicas are updated on each graph partition after execution. The update requires costly PCIe data transfer, which becomes a bottleneck. Multi-hop replication, on the other hand, alleviates the overhead of inter-GPU communication by reducing the number of times of replica update, as multi-hop replicas aren't updated after every iteration.

In Gunrock's BFS implementation; Merrill, *et al*.'s expand method [11] has been used. During the advance stage, this implementation sets a label value for each vertex to show the distance from the source, and/or sets a predecessor value for each vertex that shows the predecessor vertex's ID. The base implementation uses atomics during advance to prevent concurrent vertex discovery. When a vertex is uniquely discovered, its label (depth) and/or predecessor ID is set. Gunrock's fastest BFS uses the idempotent advance operator, thus avoiding the cost of atomics and uses heuristics within its filter that reduce the concurrent discovery of child nodes.

B. Distributed Breadth First Search

In graph algorithms, the CPU execution time is only a small fraction of the overall execution time due to low arithmetic intensity. While dealing with a distributed environment, data might reside in a remote memory location, spending a large proportion of time in sending and receiving data. Furthermore, graph algorithms do not have a regular communication pattern; messages' size and the number of sending and receiving parties vary throughout their execution. It can be rightly stated that the bottleneck of a distributed BFS is the communication. The optimization of the communication among tasks is crucial for an efficient BFS algorithm on a distributed architecture [2].

In a distributed cluster of GPUs, it is not possible to use an algorithm based on the static mapping of vertices on arrays (adjacency lists). The current and the next level frontier must be an array of exactly |V| elements. Then, a trivial static mapping makes use of a thread for each vertex in the graph. But for large graphs, the number of vertices |V| might be too high to store a global array of size |V| in the memory of a single node. In distributed environments, vertices are scattered among multiple nodes and each node only holds a subset of the whole graph. The number of edges assigned to tasks need be balanced.

P. Harish [12] and S. Hong [5] use the aforementioned static mapping. The authors in [4] use a global bitmask array to mark visited vertices which reduces the size of the global masking array to great extent but this solution is not much scalable. The maximum size of the graph that could be processed will be limited by the maximum size of the array that fits the memory of GPU.

All the shared memory optimizations speed up the visit of local vertices. In the distributed problem, however, the time spent to execute this operation is only a small fraction of the total running time which is dominated by the part of the algorithm that copes with non-local vertices [2].

The multi-node multi-GPU implementations of BFS have been presented by E Mastrostefano, *et al*. [2] and SYSTAP's Blazegraph, both of which are not open source. Therefore, a DiGPU implementation has been proposed to perform distributed graph traversal using BFS. The next section describes the design and implementation of the proposed graph traversal algorithm.

III. METHODOLOGY AND IMPLEMENTATION

The purpose of this section is to elaborate the design and implementation of DiGPU. DiGPU is a flexible user friendly CUDA-based implementation of BFS, which can be executed both on a single node and a multi-node system. For this purpose, it incorporates Unified Virtual Access (UVA) between host and device, and Peer-to-Peer direct access between devices. The BFS algorithm has been implemented on single-node single GPU system as well as on single node multi-GPU systems.

A. Module Overview

DiGPU provides following features to the user shown in Fig. 1:

1) *_SetDataFilePath*, to take the complete path of the location of graph data set.

2) *_CreateGraph*, to read and parse graph dataset and create adjacency list from it.

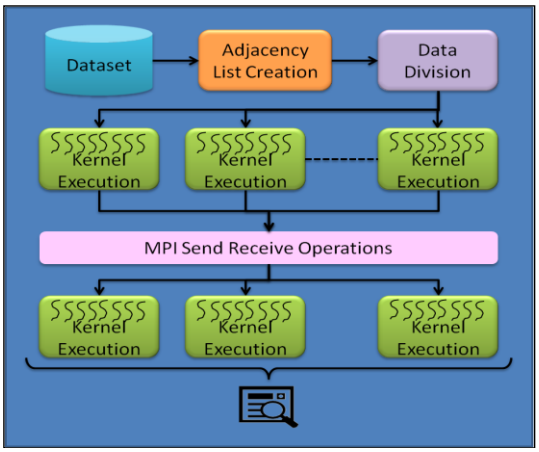

Fig. 1. Generic flow of processing in the DiGPU API

3) _ClustScan, *to examine the cluster and propose the suitable launching configurations.*

4) *_SetConfigs*, to set system information, how many nodes have been connected and how many devices every node has.

5) *_BFS*, to perform breadth first search on the created graph, with randomly selected node as root node.

_BFS utilize the following functions to perform intended operations:

a) Queue operations maintaining consistency through atomic operations, *_enqueue*, *_dequeue*, *_copyQueue*, and *_mergeQueue.*

b) *mapThreads* maps created threads to the nodes in next level frontier set. This also handles the load distribution part.

c) *CUDA Kernels*.

d) *_send* & *_receive* are the functions for data exchange among nodes using MPI.

Fig. 2 illuminates the generic flow of the API core functions. Dataset is read and parsed and then converted into an offset array and adjacency list shown in Fig. 3. The offset array and adjacency list are divided among available devices and kernels (repeatedly). The results are then combined and presented. The flow of distributed BFS in DiGPU presented in Fig. 2 also explains the order of execution of functions to accomplish BFS traversal of a graph.

B. Data Structure and Graph Representation

The adjacency list representation of graph has been used in our implementation, as there is a constraint of limited memory while working with GPUs, adjacency list is the better option than adjacency graph as it consumes lesser space.

For a graph G = (V, E), an array 'Va' of size 2|V| is used to store vertices of the graph, whereas, another array 'Ea' of |E| elements is used to hold the edges. Provided i = 0 …. |V|-1, 2Va[i] represents the offset of neighbors of node i in the Ea array and 2Va[i]+1 represents the number of neighbors of node i. Where, node b is the neighbor of node 'a' if there is a directed edge from 'a' to 'b'.

Fig. 3 represents the graph structure used in our proposed system. The offset array holds the nodes of graph in form of pairs, one index represents the index where the first neighbor of a node is located in the adjacency list and the second one is the count of neighbors of that node. For instance, in Fig. 3, node 1 has its first neighbor at 2nd index, and the 3 following it shows that it has a total of three neighbor nodes.

C. Parallel Breadth First Search – Single Node

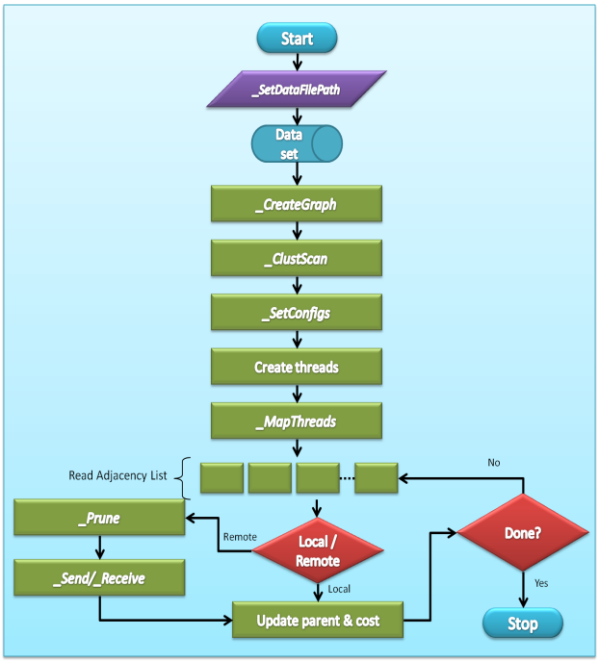

Fig. 2. Flow Diagram of Distributed BFS in DiGPU

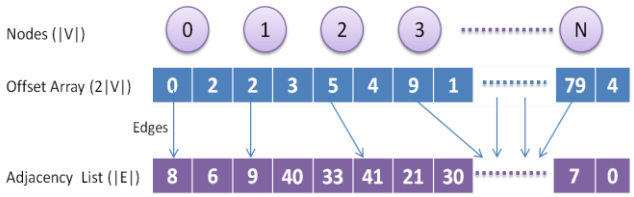

Fig. 3. Adjacency list representation of graph data

TABLE I. P2P ACCESS STATUS OF GPU DEVICES IN CDER CLUSTER

From	To	Access Allowed
GPU0	GPU1	Yes
GPU1	GPU0	Yes
GPU2	GPU3	Yes
GPU3	GPU2	Yes

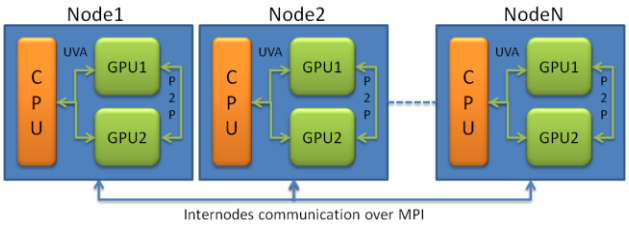

Fig. 4. Representation of Distributed Multi-GPU Cluster

The algorithm proposed in [12] has been extended for multiple GPUs on a single node. The algorithm requires one thread to be created in order to process one vertex. Two arrays namely, frontier Fa and visited Xa each of size |V| are used to store the BFS frontier and the nodes that have been visited vertices. An array to stores the minimal number of edges of each vertex from the source vertex S, the cost array Ca is also used. In every iteration, each vertex checks its entry in the frontier array Fa. If it is positive, it fetches its cost from the cost array Ca and updates all the costs of its neighbors if more than its own cost plus one using the edge list Ea. The vertex removes its own entry from the frontier array Fa and adds to the visited array Xa. It also adds its neighbors to the frontier array if the neighbor is not already visited. This process is repeated until the frontier is empty. The algorithm terminates when all the levels of the graph are traversed and frontier array is empty.

This implementation has also been extended for shared-memory multi-GPUs, utilizing the CUDA Peer-to-Peer (P2P) Direct Access (Table 1). The offset array was divided equally amongst the GPU devices on the node (which enable P2P access with other devices). The adjacency list is then divided such that each device gets the complete set of those nodes which are adjacent to the nodes assigned to it.

Host keeps track of the allocations and utilizing the Unified Virtual Addressing facilitated by CUDA the devices could know which device has their desired node if they are missing any, and then using P2P access that device gets a copy of it.

D. Distributed Multi-GPU Approach

The distributed graph traversal has been implemented on multi-GPU cluster, following and eventually extending the work presented by Mastrostefano, *et al*. [10] towards a hybrid model. The cluster setup represented in Fig. 4 shows that it has multiple nodes, each node having more than one GPU devices. The system perform as a shared memory implementation within a node, using Peer-to-Peer direct access when devices are communicating with each other and Unified Virtual Addressing while host and device are communicating. The system follow distributed approach across the nodes, i.e., communication using *_send* and *_receive* functions.

IV. Experimental Setup

The experimental setups have been used to perform the experimentation of the proposed DiGPU system. Details of these setups have been explained as follows:

A. Cluster Setups

Fours clusters have been used to perform our experiments for DiGPU. The details of hardware specifications of utilized clusters are enlisted below:

1) NVIDIA Research Center (SysLab) Cluster – FAST NUCES

This cluster consisting of following devices have been set up specifically for our experimentation purpose shown in Table 2:

- Standalone PCs with:
  - NVIDIA Tesla K40 and GTX-780
  - NVIDIA GTX-780 and GTX-780
  - NVIDIA Tesla K40 and NVIDIA Tesla
- Multi-node cluster with the nodes listed above.

2) DER Cluster – Georgia State University

This cluster has 19 nodes out of which three nodes have CUDA supported GPU cards:

- GPU07 – 1 GeForce GTX TITAN Black
- GPU11 – 4 GeForce GTX 770
- GPU10 – 4 Tesla K20c.

Above nodes have:

- Processors Dual Intel Xeon E5-2650 with 64 GB memory
- Operating system CentOS 6.7.
- CUDA 7.5

TABLE II. Configurations of Clusters Involved in Experiments

Configuration	Nodes	GPUs/node	GPU Specifications	P2P Access Status
A	1	4	4 GeForce GTX 770	Refer Table 1
B	1	4	4 Tesla K20	Refer Table 1
C	2	2	1 Tesla K40c 1 GeForce GTX 780	Enabled between devices of 1 node
D	1	2	1 Tesla Kxxx 1 GeForce GTX 7xx	Enabled

TABLE III. BREADTH FIRST SEARCH EXECUTION TIMES

Experimental Setup	Vertices	Edges	Processing Time (sec)
Single Node – Single GPU	2394385	5021410	1635.9
Single Node – 2 GPUs	2394385	5021410	1089.3

TABLE IV. DISTRIBUTED BFS USING HYBRID MODEL EXECUTION TIMES

Experimental Setup	Dataset	Processing Time (sec)
CDER cluster – Configuration A	WikiTalk ~ $2.3x10^6$ vertices	349.325377
	Synthetic Graph – $10x10^6$ vertices	772.945530
CDER cluster – Configuration B	WikiTalk ~ $2.3x10^6$ vertices	363.795705
	Synthetic Graph – $10x10^6$ vertices	607.428377
SysLab cluster – Configuration C	WikiTalk ~ $2.3x10^6$ vertices	849.862036
	Synthetic Graph – $10x10^6$ vertices	1809.419021
CDER cluster – Configuration D	WikiTalk ~ $2.3x10^6$ vertices	1003.844756
	Synthetic Graph – $10x10^6$ vertices	4346.329110

3) Cloud Intances

For our preliminary results, two categories of Amazon Web Services EC2 GPU instances have been used with following specifications:

- High Frequency Intel Xeon E5-2670 (Sandy Bridge) Processor
- NVIDIA Grid GPUs, each with 1,536 CUDA cores and 4GB of video memory
- Ubuntu LTS 14.04
- CUDA 7.0

B. Datasets

Both synthetic datasets and real-world graph have been used to experiment in this research. PaRMAT [13] is used to generate synthetic graphs of various sizes to perform impact analysis of load distribution based on GPU specifications. Experiments have been performed on graphs as large as 10M vertices.

WikiTalk dataset from SNAP [14] has been used as a real-world graph to test the implementation. Each registered user of Wikipedia has a talk page, which could be edited by either them or other users to communicate and discuss updates to various articles on Wikipedia. In this SNAP WikiTalk dataset, there is information extracted from all user talk page changes and converted in the form of a network.

The network contains all the users and discussion from the inception of Wikipedia till January 2008. Nodes in the network represent Wikipedia users and a directed edge from node i to node j represents that user i at least once edited a talk page of user j. The dataset represents a directed graph, with 2,394,385 nodes and 5,021,410 edges [14].

V. RESULTS

This section explains the outcomes of the testing of proposed Hybrid Model for DiGPU and also presents the implications of proposed load-balancing based of specifications of GPUs.

A. Preliminary Results – PRAM Algorithm Implementation

The results in Table 3 are preliminary results which had been obtained using a primitive single node PRAM implementation of BFS using Harish's algorithm [12]. The results had been computed using Amazon AWS instances on the WikiTalk graph dataset.

B. Distributed BFS – Hybrid Model Implementation

Hybrid model has been tested on several datasets for all four experimentation setups, the major and important results are those which have been obtained from the 10M vertices synthetic graph and WikiTalk dataset. These results have been summarized in Table 4.

Traversed Edges per Second (TEPS) is a metric well known to measure performance related to graph operations. The results of execution of DiGPU hybrid model on local SysLab cluster (Configuration C) have been represented in the form of TEPS in Table 5. The TEPS for each graph traversal instance in Table 4 show improvement with increase in number of vertices in the graph. This depicts that even though there is network latency issue DIGPU's is scalable for large graphs and the given set of results show improvement in performance with increase in graph size.

Fig. 5 is the visual representation of the comparison amongst the four experimentation benches, in terms of Thousand TEPS. Setup A, B, C all three have four GPU devices but Fig. 5 clearly indicates that Setup C lags behind both Setup A and B. The reason is network latency introduced by the switch used to connect nodes at SysLab. An improvement might be observed if there would have been an Infiniband switch. Also, on test bed Setup C, only one node offers P2P access between the two GPUs in it.

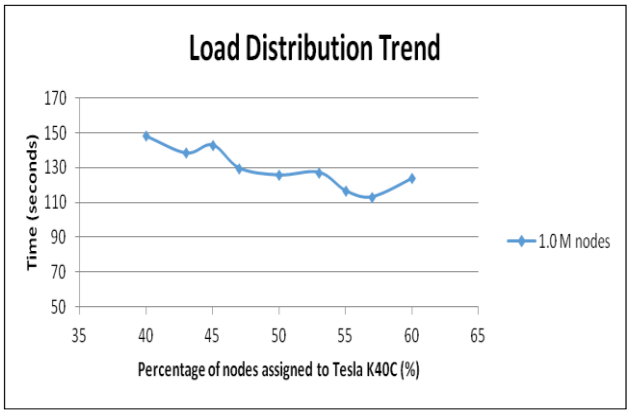

Fig. 5. . Load Distribution Trend Synthetic Graph of 1 million nodes

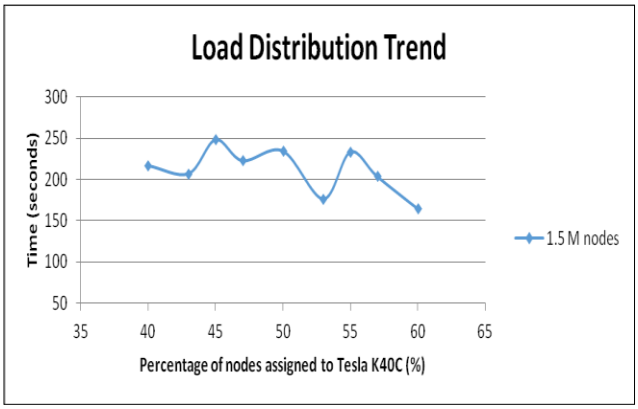

Fig. 6. Load Distribution Trend Synthetic Graph of 1.5 million nodes

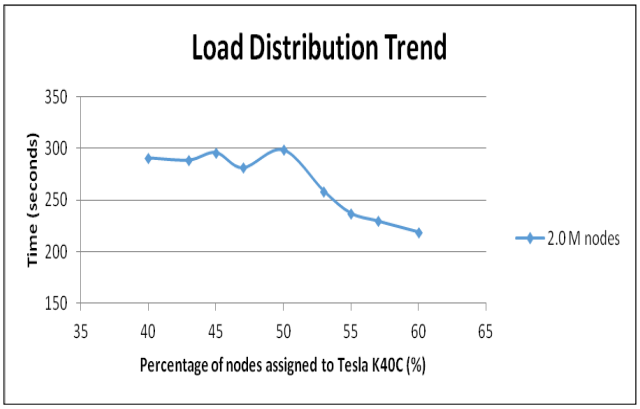

Fig. 7. Load Distribution Trend Synthetic Graph of 2 million nodes

TABLE V. PERFORMANCE OF HYBRID MODEL ON LOCAL CLUSTER WITH VARYING DATA SIZES

Dataset	Processing Time (sec)	Traversed Edges Per Second
Synthetic Graph – $1.0x10^6$ nodes	206.004216	4854.269584
Synthetic Graph – $1.5x10^6$ nodes	274.754027	5459.428625
Synthetic Graph – $2.0x10^6$ nodes	413.738531	4833.970853
Synthetic Graph – $2.5x10^6$ nodes	533.307219	4687.729531
Synthetic Graph – $3.0x10^6$ nodes	595.089924	5041.254907
Synthetic Graph – $10x10^6$ nodes	1809.419021	5526.635833

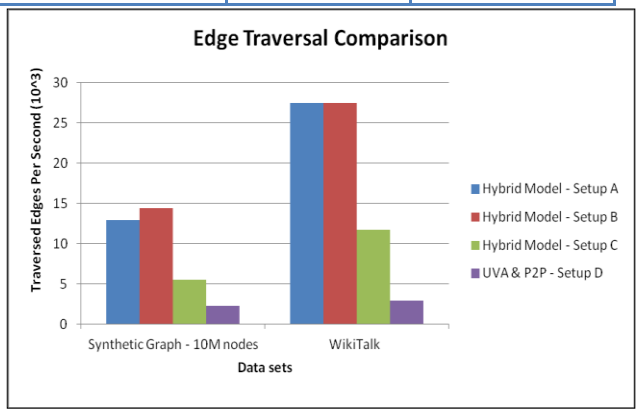

Fig. 8. TEPS Comparison of Experimental Setups

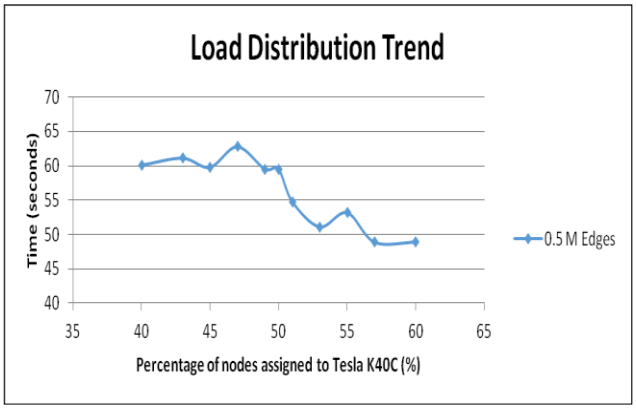

Fig. 9. Load Distribution Trend Synthetic Graph of 0.5M nodes

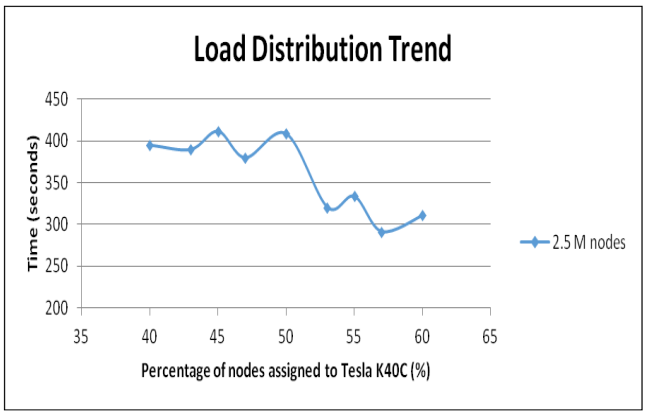

Fig. 10. Load Distribution Trend Synthetic Graph of 2.5 million nodes

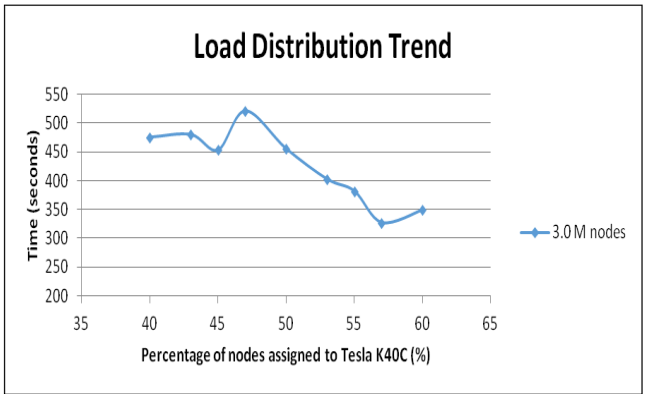

Fig. 11. Load Distribution Trend Synthetic Graph of 3 million nodes

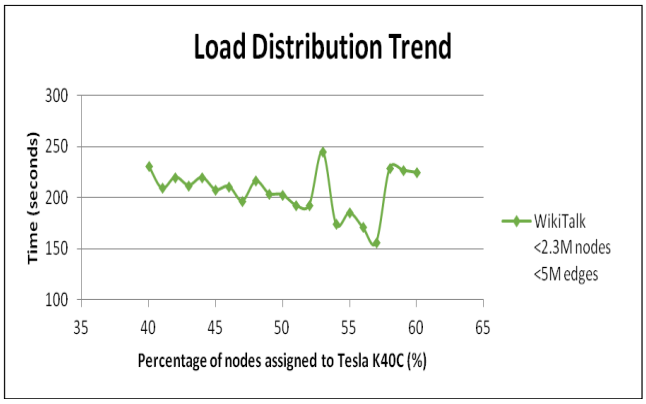

Fig. 12. Load Distribution Trend of Real World graph of WikiTalk

C. Load Distribution

Load distribution experiments have been performed on six synthetic graphs of varying sizes on SysLab systems having Tesla K40c and GeForce GTX 780 GPUs.

Graphs in Fig. 6 to 12 show that execution time improved when the distribution of load was in favor of Tesla device. This is due to the fact that Tesla is better device with more computation power, better memory bandwidth, more CUDA cores and larger memory when compared to GeForce GTX 780 device.

Load distribution using GPU specification represents improvement in execution time of graph traversal. Performance of Hybrid model is better on CDER cluster than SysLab setups which is obviously due to network latencies [15].

VI. Conclusion and Future Work

DiGPU is a software module for multi-node, distributed cluster graph traversal in form of Breadth First Search on GPUs. Results show that our proposed Hybrid Model for graph traversal in multi-GPU clusters performs better than its PRAM and single node UVA-P2P counterpart. Though UVA-P2P on a single node is the base case of the hybrid model its performance is greatly affected by infrastructure, for instance the bandwidth of interconnects between the GPUs and also the processor in the node. The bottleneck in performance is network latency which is inherent in a distributed cluster. But for a cluster with a better interconnect between its nodes or when there are multiple GPUs on a node amongst which not all are paired with each other, our Hybrid Model implementation works better, and our results from experiments performed on CDER cluster clearly support that.

Our motivation behind this research was to explore aspects of improvement in graph analysis, the experiments performed to study the impact of GPU specifications also show a trend in favour of the better GPU device when more nodes are allocated to it, and after a certain increment in load distribution ratio the results begin to deteriorate which is explainable as there would definitely be more communication due to large imbalance in communication.

Acknowledgment

This research has been supported by NVIDIA Teaching and Research Center awards. The authors also acknowledge the Georgia State University, for providing us the testing environment. The authors would like to thank Mr. Muhammad Rafi, and Mr. Ali Ahmed for their valuable input.

References

[1] Lumsdaine, Andrew, et al. "Challenges in parallel graph processing." Parallel Processing Letters 17.01 (2007): 5-20.

[2] Vibhav Vineet and P. J. Narayanan,“Large graph algorithms for massively multithreaded architecture” in Proc. HiPC, 2009.

[3] S. Kumar, A. Misra, R. S. Tomar,“A Modified Parallel Approach to Single Source Shortest Path Problem for Massively Dense Graphs Using CUDA”, Int. Conf. on Computer & Comm. Tech. (ICCCT), IEEE, 2011.

[4] L. Luo, M. Wong, and W.-M. Hwu, An Effective GPU Implementation of Breadth-First Search, in Proc. DAC, 2010, pp. 52-55.

[5] S. Hong, S.K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating CUDA Graph Algorithms at Maximum Warp,” in Proc. PPoPP, 2011, pp. 267-276

[6] Hong, Sungpack, Tayo Oguntebi, and Kunle Olukotun. "Efficient parallel graph exploration on multi-core CPU and GPU." Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on. IEEE, 2011.

[7] Merrill, Duane, Michael Garland, and Andrew Grimshaw. "Scalable GPU graph traversal." ACM SIGPLAN Notices. Vol. 47. No. 8. ACM, 2012.

[8] A. Grimshaw, D. Merrill, M. Garland, “High Performance and Scalable GPU Graph Traversal”, Technical Report, Nvidia, 2011.

[9] Wang, Yangzihao, et al. "Gunrock: A high-performance graph processing library on the GPU." Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2015.

[10] Mastrostefano, Enrico, and Massimo Bernaschi. "Efficient breadth first search on multi-GPU systems." Journal of Parallel and Distributed Computing 73.9 (2013): 1292-1305.

[11] Leskovec, Jure, and Andrej Krevl. "{SNAP Datasets}:{Stanford} Large Network Dataset Collection." (2014).

[12] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA”, High Performance Computing – HiPC 2007, S. Aluru, M. Parashar et al. (Eds.), Springer Berlin Heidelberg 2007, pp. 197-208.

[13] Khorasani, Farzad and Vora, Keval and Gupta, Rajiv, " PaRMAT: A Parallel Generator for Large R-MAT Graphs", Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques, 2015.

[14] Bernaschi, Massimo, et al. "Enhanced GPU-based distributed breadth first search." Proceedings of the 12th ACM International Conference on Computing Frontiers. ACM, 2015.

[15] Durrani, Muhammad Nouman, and Jawwad A. Shamsi. "Volunteer computing: requirements, challenges, and solutions." Journal of Network and Computer Applications 39 (2014): 369-380.