cuSTINGER: Supporting Dynamic Graph Algorithms for GPUs

Oded Green, David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA

***Abstract*— cuSTINGER, a new graph data structure targeting NVIDIA GPUs is designed for streaming graphs that evolve over time. cuSTINGER enables algorithm designers greater productivity and efficiency for implementing GPU-based analytics, relieving programmers of managing memory and data placement. In comparison with static graph data structures, which may require transferring the entire graph back and forth between the device and the host memories for each update or require reconstruction on the device, cuSTINGER only requires transferring the updates themselves; reducing the total amount of data transferred. cuSTINGER gives users the flexibility, based on application needs, to update the graph one edge at a time or through batch updates. cuSTINGER supports extremely high update rates, over 1 million updates per second for mid-size batched with 10k updates and 10 million updates per second for large batches with millions of updates.**

I. Introduction

Dynamic graphs are ubiquitous and are used to represent data sets across various application domains. For example, dynamic graphs are used for representing the ever-changing relationship between players in social networks or for representing transactions between entities in financial or communication networks. Dynamic networks can also be used for representing protein interactions in a biological network.

Static graph algorithms can be used to analyze dynamic graphs. One approach for dynamic graph analytics is to run a static graph algorithm after each update to the graph. This approach is often computationally expensive and infeasible for large graphs.

Dynamic graph algorithms focus on analyzing graphs that are constantly changing. By using some previous state, dynamic graph algorithms can typically avoid a full out recomputation of the analytic. Because of this, dynamic graph algorithms can be orders of magnitude faster than their static graph counterparts and can enable online monitoring of the network. Lastly, dynamic graph algorithms typically require advanced data structures for both the algorithm's data and for the graph representation.

In this paper we present *cuSTINGER*, the first data structure for maintaining dynamic graphs for NVIDIA's CUDA supported GPUs. *cuSTINGER* manages the allocation of memory on the GPU for the dynamic graph alleviating the programmer's need to create a dynamic graph data structure and allowing the algorithm designer to focus purely on the algorithm. *cuSTINGER* is a GPU extension of the STINGER (see [3], [10]) data structure used for representing dynamic graphs. STINGER has enabled the development of a large number of dynamic graph algorithms, including online monitoring of betweenness centrality [12], connected components [16], and counting and tracking the number of triangles [9]. The design of cuSTINGER exploits the architecture of the GPU. In this paper, we differentiate between the STINGER data structure as defined in STINGER [3] and an implementation of STINGER as given by Georgia Tech [10] (which we call GT-STINGER).

cuSTINGER can by configured at runtime by the programmer or analyst. These includes enabling certain properties within the data structure. This includes adding support for weighted or property graphs, where vertices and edges have an associated type reflecting a possible role or relationship in the network. As the memory on the GPU is a more limited resource in comparison with the CPU, *cuSTINGER* is designed to allow users to control the needed features and the total amount of allocated-memory; meeting the application and network requirements. While *cuSTINGER* is designed to support streaming graph analytics, it still permits static graph algorithms. In this paper, we compare the performance of static graph triangle counting [13] with a cuSTINGER implementation. There is a slight reduction in performance between these two implementation, where cuSTINGER is $1\% - 10\%$ slower. This supports our intuition that our new data structure works well with static graph algorithms with negligible overhead.

cuSTINGER has several different built-in memory allocators for controlling the amount of memory allocated for each vertex. For example, when *cuSTINGER* is used for a static graph algorithm it can allocate the exact amount of memory required for the static graph. For dynamic graph algorithms cuSTINGER can use different allocators that trades memory utilization for improved performance.

II. Related Work

A. Dynamic Graph Data Structures

The nature of dynamic networks requires that the data structure used for maintaining the network be flexible enough to support the insertion and deletions of edges and vertices. This requirement limits the ability to use data structures designed for static networks. For example, the Compressed Sparse Row (CSR) representation, used for both graph problems and sparse linear algebra problems, compacts the data into a handful of compressed arrays such that the exact amount of memory needed for representing the graph is allocated. The tight bound on memory allocations limits

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

the use of CSR for dynamic graph algorithms as each update to the graph would require creating a new CSR representation, leading to a significant overhead. An adjacency matrix requires a matrix of $|V| \times |V|$ elements. While this representation over allocates the amount of memory needed to represent most real-world and sparse networks, it allows for updating the graph in constant time with relative ease. Linked lists also offer the flexibility to add and delete elements with a relative ease while offering tighter memory bounds than the adjacency matrix. However, finding an element requires more memory references as this requires a traversal of the entire linked list. This is extremely inefficient given that each element of a linked list may be located in a non-contiguous space in memory. CSR offers more efficient lookups as the adjacency lists are located in contiguous memory.

STINGER [3] is purposely designed for efficient implementations of dynamic graph algorithms. STINGER takes the best features of an adjacency matrix, linked lists, and CSR, and uses a mix of these methods to give extremely fast updates while also allowing for the efficient implementation of various analytics. STINGER uses a blocked linked list data structure that stores multiple edges in each block of the linked list. These are referred to as edge blocks. In the GT-STINGER there is an internal memory manager for allocating edge blocks to the vertices. By doing so, this implementation is able to avoid using system calls for memory allocation and deallocation. Such systems calls can require thousands of cycles and may add significant overhead. The edge block size is a user controlled parameter. A small edge block size results in the same restrictions of a linked list whilst a large edge block has the same drawback of an adjacency matrix (over allocating memory without the efficient lookups). Thus, selecting the edge block size is a compromise between algorithmic efficiency and storage efficiency and should be decided based on the application's requirements.

In [10] it is shown that GT-STINGER can ingest several million updates per second on a single shared-memory system. In [15], GT-STINGER is compared to several leading graph databases and libraries, including shared and distributed-memory. GT-STINGER is able to outperform these databases both in terms of update rates and for finding the connected components for a graph. A distributed version of STINGER, called DISTINGER, is introduced by Fong *et al.* [11]. DISTINGER uses a hash-map to decide the compute-node where a vertex will be stored. While DISTINGER is able to handle larger graphs due to its distributed nature, including additional memory, it still suffers from having one centralized server that is responsible for updating the entire network.

Lastly, LLama [14] offers an alternative data structure for dynamic graph algorithms that is based on check-pointing the network as updates are added. An adjacency list of a vertex might be split across multiple checkpoints in LLama. This makes the LLama data structure an unlikely candidate for the GPU as accessing an edge in each check will cause low system utilization.

B. GPU Graph Analytics and Platforms

In [19], Merrill *et al.* give the first scalable GPU graph traversal for NVIDIA's CUDA [1] supported GPU. To improve the utilization of the GPU, several different traversal strategies are taken. These strategies take into the account the massive thread-level parallelism available for CUDA supported GPU cards. Thus, large vertices are processed differently than smaller vertices.

Other available graph analytics for the GPU include triangle counting [13], [22], connected components [24], single-source shortest path [8], betweenness centrality for static graphs [23], [18], betweenness centrality for dynamic graphs [17], and community detection [25].

In addition to these implementations, several libraries for static graph analytics for the GPU include: Gunrock [27], [21], BlazeGraph [5], nvGraph [20], GasCL [6], and BelRed [7]. These libraries offer data scientists simple tools needed for analyzing data-sets without the need to write complex GPU code. Gunrock [27] is designed for implementing graph traversals through the use of a small set of highly optimized operators which are suitable for a wide range of applications. BlazeGraph [5] has its own domain-specific language, called DASL, enabling programmers to implement advanced analytics with high-level functionality. BelRed [7] allows programmers to implement graph based applications using basic building linear algebra building blocks. GasCL [6] uses a vertex-centric approach approach with the Gather-Apply-Scatter (GAS) model. In this paper we show that for static graph analytics, cuSTINGER adds negligible overhead.

III. cuSTINGER

In this section we outline our design and implementation of cuSTINGER. We discuss data structure considerations as well as performance trade-offs. When relevant we compare our cuSTINGER with GT-STINGER.

A. Structure of Array Vs. Array of Structure

In GT-STINGER, edges are stored in edge blocks, where each block contains multiple edges. Each edge stores associated properties such as when that edge was last modified or its weight. These are stored in a structure; therefore, the edge block is an array of structures. Such a representation is not ideal for CUDA supported GPUs as this can increase the number of memory requests for updating a single field across multiple threads. A more efficient memory access pattern for CUDA would be to have the same fields accessed in a consecutive fashion. As such, in cuSTINGER, we replace the array of structures with a structure of arrays allowing for a reduced number of memory requests. An additional benefit of this modification to the data representation is our ability to support different allocation modes for using cuSTINGER (Section III-E).

B. Replacing Edge Blocks with Arrays

In cuSTINGER we replace the use of the list of edge blocks per vertex with a single adjacency array for each vertex. In GT-STINGER there is a memory manager that is

TABLE I

NETWORKS USED IN OUR EXPERIMENTS: 10TH DIMACS GRAPH IMPLEMENTATION CHALLENGE [4] AND SNAP [2]. $|E|$ REFERS TO DIRECTED EDGES. NOTE THAT NETWORKS ARE ORDER BASED ON THE NUMBER OF VERTICES (ASCENDING ORDER). TIME IS GIVEN IN SECONDS FOR THE INITIALIZATION AND TRIANGLE COUNTING. TC-CSR IS THE EXECUTION TIME FOR THE TRIANGLE COUNTING IMPLEMENTATION USING CSR AND TC-CUS FOR THE IMPLEMENTATION USING CUSTINGER.

Name	Network Type	$\|V\|$	$\|E\|$	Ref.	Init. (sec.)	TC-CSR (sec.)	TC-cuS (sec.)
prefAttach	Powerlaw	100k	1M	[4]	0.110	0.108	0.110
m14b	Walshaw	215k	3.3M	[4]	0.24	0.240	0.260
coAuthorsDBLP	Social	299k	1.95M	[4]	0.33	0.218	0.242
amazon0601	Social	403k	3.38M	[2]	0.45	1.187	1.253
web-BerkStan	Webcrawl	685k	7.6M	[2]	0.81	0.706	0.780
audikw1	Matrix	943k	76.7M	[4]	1.25	4.485	4.682
ldoor	Matrix	952k	45.5M	[4]	1.21	2.674	2.916
as-skitter	Trace route	1.69M	11.1M	[2]	2.12	57.14	59.37
kron_g500-logn21	Kronecker	2M	201M	[4]	3.03	2992.7	2996.5
cit-Patents	Citation	3.77M	16.5M	[2]	4.83	0.814	0.830
soc-LiveJournal1	Social	4.84M	68.99M	[2]	7.07	8.223	8.767
cage15	Matrix	5.15M	94M	[4]	7.14	6.544	7.204
road_central	Road	14M	33M	[4]	19.9	7.524	8.581
uk-2002	Webcrawl	18.52M	523M	[4]	28.4	424.9	431.4

responsible for edge block allocation and deallocation from a pre-allocated global memory pool. Unfortunately, the edge block approach is not applicable to the GPU architecture. First of all, edge blocks might be located in different places in the memory system, increasing the number of memory accesses and reducing the benefits of caching and pre-fetching. Also, if a small edge block size is used (which is preferable from a storage perspective), then a GPU's streaming multiprocessor migh be underutilized as there is not enough work to use a full GPU warp. To overcome this work utilization issue it is possible to use large edge blocks. However, this leads to low storage utilization, which is an already valuable commodity on the GPU. To avoid the above, we allocate a single array for each adjacency list. This has the benefit of better locality for edges in the same adjacency list, better memory utilization, and requires only one memory allocation (allowing for a simpler memory manager).

C. Data Layout and Memory Management

In cuSTINGER, we implement an advanced memory manager that requires fewer calls to the system memory allocation and deallocation functions than a naive implementation. Specifically, numerous adjacency lists are grouped together into one larger chunk of contiguous memory. This improved memory manager reduces the time spent in the initialization phase, by well over an order of magnitude. Further, updating the graph also becomes significantly faster.

D. Memory Allocation Modes

While cuSTINGER has been designed for developing dynamic graph algorithms for the GPU, it can also be used for static graph algorithms. As such, cuSTINGER allows the programmer to select a memory allocation model that is preferable to the application at hand. cuSTINGER has memory allocation modes for both static and dynamic graphs. For static graph algorithms operating on static graphs, cuSTINGER can allocate the exact amount of needed memory - similar to the memory requirements of CSR. This is known as the *exact* mode.

Algorithm 1: Pseudo code for updating a cuSTINGER graph with a set of B edge insertions. Functions followed by $<<<>>>$ denote a call to a GPU kernel.

Inputs: $cuSting$ - dynamic graph.
Inputs: B - batch of updates.
$CopyHostToDevice(B)$
$Phase1Update$ $<<<>>>$ $(cuSting, B, |B|)$// GPU kernel

$CopyDeviceToHost(\hat{B})$ // Uncompleted edge list
$CopyDeviceToHost(Dups_B)$ // Duplicated inserted edges in B
$RemoveDuplicates$ $<<<>>>$ $(Dups_B)$ // GPU kernel

// Reallocates memory for all unique sources vertices of $\hat{B}$.
$CopyDeviceToHost(PointerAdjacencyList)$
CPU-ReallocateMemory($\hat{B}, PointerAdjacencyList$)
$CopyHostToDevice(NewPointerAdjacencyList)$

// Copies entire edge lists for all vertices new edge lists
$DeepCopyDeviceToDevice(PointerEdgeList, NewPointer-EdgeList, \hat{B})$
$Phase2Update$ $<<<>>>$ $(cuSting, \hat{B}, \hat{B})$// GPU kernel

cuSTINGER also has dynamic graph memory allocation modes. These trade off storage utilization and the ability to support updates with as few calls to the system memory allocation functions. Such calls are relatively expensive in comparison to the update operation. Users are responsible for selecting their preferred memory allocation mode as part of their initial configuration.

E. cuSTINGER Meta-Data Modes

Current state of the art GPUs have a smaller amount of memory in comparison with their CPU counterparts. For example, a single NVIDIA K40 GPU has 12 GBs of memory and the K80 has 2×12GBs memory. Currently CPU servers can support tera-bytes of memory in a single system. As such, cuSTINGER has several different meta-data modes that can be used in the initialization. These modes are intended for different types of applications requiring additional properties to the graph:

1) Adjacency-only mode - this is a stripped down network that only stores the adjacency lists and is for unweighted networks. Conceptually, this is similar to the amount of data stored in CSR.

2) Vertex weights and edge weights - a user can decide to assign a weight to vertices or edges. The additional storage cost is $O(V)$ for vertex weights and $O(E)$ for edge weights.

3) Semantic mode - is intended for analysts that need to associate additional properties for each vertex or edge. This includes vertex types, edge types, and time stamps for when the edge was last modified. time stamps are especially useful for monitoring temporal networks. Timestamps were added to cuSTINGER to support portability with STINGER. Lastly, semantic mode also allocates memory for vertex and edge weights. In total, semantic mode increases the memory footprint by $O(V + E)$.

These modes allow for better control over the amount of allocated memory. This is especially useful for analytics and applications that do not require additional data. In GT-STINGER the semantic support is turned on by default and in fact cannot be removed due to the use of array of structures

rather than structures of arrays. cuSTINGER is able to support these additional modes partially due to the change made in its internal representation.

F. Updates

cuSTINGER supports the following operations to the graph: edge insertions, edge deletions, vertex insertions, and vertex deletions. Given the massive parallelism on the GPU, it is highly recommended that updates be grouped into batches (as is done in [10]). However, the decision on the update granularity is typically application driven. For example, in dynamic graph betweenness centrality [12] the update-time is negligible in comparison to the computational requirement, thus, the granularity is not important. However for dynamic graph algorithms such as triangle counting [9] and tracking connected components [16] the overhead can become dominant.

cuSTINGER separates the insertion and deletion processes, unlike GT-STINGER which can do both of these concurrently. This is partially due to the increased parallelization of a GPU and the more complex memory management. By separating the insertions and deletions, the actual update implementation is simple. In the following subsections, we discuss these update processes in additional detail. This discussion focuses on unweighted and non-semantic graphs for simplicity. For the next section we assume that a batch of updates, B, consists of either $|B|$ edge insertions or deletions.

1) Edge insertions: The pseudo code in Alg. 1 depicts the edge insertion process for cuSTINGER. Initially, the batch is copied to the device, where cuSTINGER is located. This is followed by the first phase sweep of the insertion process. For each edge update $b \in B$, we check if that edge already exists in the graph (duplicates are typically undesirable). All non-duplicate edges are inserted, assuming sufficient memory is available. If there is not enough memory, it is added to an edge list that requires additional attention. Further, a situation can arise where there are two updates in the same batch, $b_1, b_2 \in B$ that are identical - causing the same edge to be inserted twice. This is undesirable and cuSTINGER detects these same-batch duplicates and creates a list of these updates for further processing.

Lastly, vertices for which adjacency lists are full and cannot store all the new updates are dealt with by allocating new a adjacency list. This is done via several memory copies from the device to the host followed by an additional device kernel launch. Initially, each of the vertices requiring additional memory receives a new and larger adjacency list. This is followed by copying the old adjacency list into the new one. Finally, the non-completed edges are inserted in a second phase sweep.

2) Edge deletions: Edge deletions are to some degree easier than edge insertions. This is due to the nature of the deletions. An edge deletion will always reduce the amount of necessary storage; thus, additional memory allocations and copying are never necessary. Edge deletions are also implemented using a two-phase algorithm for a given batch. In the first phase, given a batch of edge deletions, all edges in the batch are located within cuSTINGER and marked as deleted. In the second phase, the adjacency list is compressed by taking an edge at the end of a vertex's adjacency list and replacing it with the deleted edge.

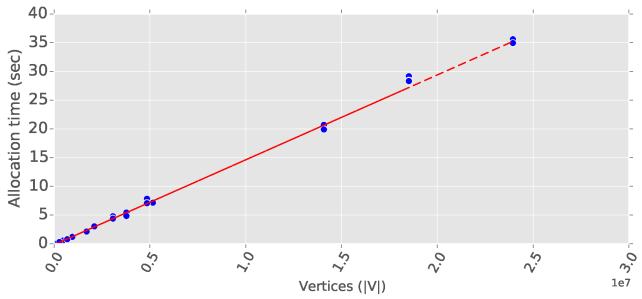

Fig. 1. Allocation time for initializing cuSTINGER as a function of the number of vertices in the initial graph. A linear trendline has been added that confirms that the initialization time grows linearly with the number of vertices.

3) Vertex insertions and deletions: Vertex insertions and deletions can be considered as a series of edge insertions and deletions, respectively. For example, if a vertex is deleted so are all of its adjacency edges. Thus, a vertex update can be completed using the edge updates operations.

IV. Performance Analysis

Our experiments are conducted on an NVIDIA K40 GPU and an Intel i7-4770K quad-core CPU system. The K40 is a Kepler based GPU with 15 SMs and 192 SPs per SM, for a total of 2880 SPs. The K40 has 12GB of GDDR5 memory. The Intel i7-4770K is a Haswell based processor running at 3.5 GHz with 8MB L3 cache. This system has 32GB of DDR3-1600 memory. To test the capability of cuSTINGER with graph updates we use real world graphs and networks taken from the 10th DIMACS Graph Implementation Challenge [4] and [2]. Details of these graphs can be found in Table I. Finally, to test the throughput rate of cuSTINGER, we check its ability to handle different batch update sizes. We use a wide range of batch update sizes, from a single update and all the way up to batches with ten million updates (in multiples of tens). The maximal size of the batch per network is dependent on its initial size. The batches are created using a random edge generator. Each experiment, consisting of one graph and one batch size, is conducted for ten different batches. For all experiments, the graph is read from a file and loaded onto the GPU, ensuring that the different batches are applied to the same initial graph. For each batch, we insert the edges into the graph and then we remove these edges using cuSTINGER's insertion and deletion functionality.

A. Initialization

Table I shows the time it takes to initialize a cuSTINGER structure on the GPU. This includes the memory allocation time for each of the adjacency lists, initializing the internal data structure properties on both the host and the device, and transferring of data between the host and the device. Fig. 1 depicts the allocation time for cuSTINGER as a function of the number of vertices in the graph. The solid curve denotes

the linear rate at which the allocation time is a function of the number of vertices. This curve shows a correlation between the initialization time and the number of vertices in the static graph.

B. Update Rate

One of the main design goals of cuSTINGER is to ensure that it is possible to update the graph at high rates such that the update process itself does not become the bottleneck for a streaming algorithm. To measure cuSTINGER's ability to process updates, we measure the number of updates per second that cuSTINGER can handle - this is the same metric used to measure the update capability of GT-STINGER [10]. Fig. 2 depicts the updates per second rate that cuSTINGER can achieve as a function of the batch size. The upper subfigure depicts the throughput rate for edge insertions whilst the bottom subfigure depicts the rate for edge deletions.

1) Small Batch-Sizes: The update rate for both insertions and deletions is similar for small batch sizes. cuSTINGER achieves roughly 15k updates per second when the size of the batch is a single update. This rate is approximately the same across all test-graphs. For such small batches, there are several performance restrictions. The first restriction is the kernel launch overhead which can take several microseconds and dominates the execution time for small batches. The second performance restriction is the need to copy the update from the host to the device. Lastly, the GPU is simply underutilized as there is only one update to process - this utilizes a single GPU thread-block and a single GPU streaming multi-processor out of the many available. Therefore, it is not overly surprising the throughput rate increases by almost a factor of 10 and 100 as the batch size increases to 10 and 100 edges per batch, respectively. As the batch sizes increase beyond 1000 updates, these factors become less dominant and the time spent updating the graph becomes more important.

2) Edge Insertions - Large Batches: As the batch size continues to increase, the likelihood of an edge insertion not having space in a pre-existing adjacency list also increases. When this occurs, a new adjacency list needs to be allocated and the old adjacency list is copied to the new list. This adds overhead. In Fig. 2, there are several networks that have a "performance dip" for batches of 10k and 100k updates. In all these cases, updates have caused a scenario where at least one vertex requires a new adjacency list. In the current version of cuSTINGER, the update process requires several memory copies from the device to the host in addition to initialization of some auxiliary data structures. While these memory copies are not overly demanding, in comparison with the time spent updating the network they are expensive. In future work we will look into optimizing this process.

Nonetheless, for most networks cuSTINGER is able to keep an update rate of over 1 million updates per-second for larger batches and in some cases an update over 10 million updates per-second. Note, for several of the networks the batch size is close to 50% of the size of the graph - which means that due to the update, the graph increases its size by as much as 50%. While such an experiment offers performance insights,

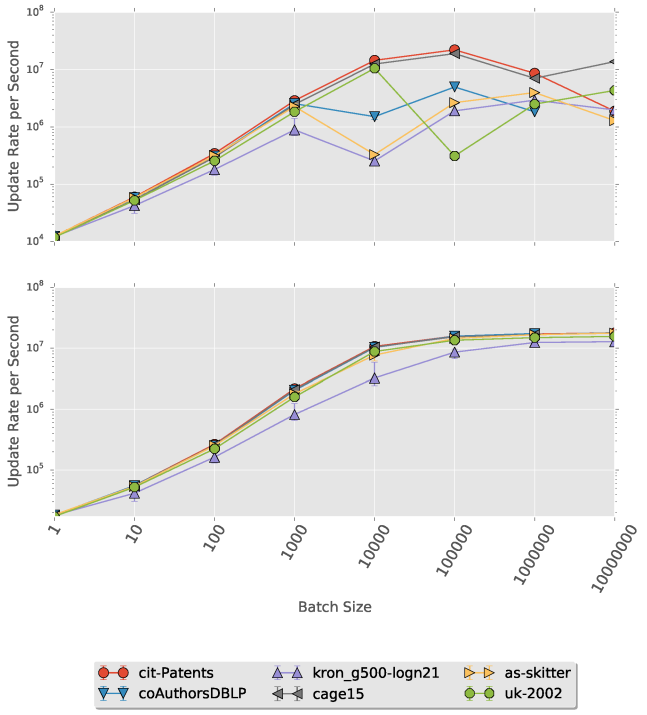

Fig. 2. cuSTINGER's update rate as a function of the batch size - higher rates are desirable. Top subfigure depicts the rate for edge insertions and the bottom subfigure for edge deletions. For small batch sizes, the update rate is limited by the kernel launch overhead.

for most dynamic networks we don't expect an update rate that will change the graph by that amount.

3) Edge Deletions - Large Batches: Unlike edge insertions that may require additional memory allocations, edge deletions do not. Edge deletions only remove existing edges from the graph. In this current version of cuSTINGER, empty edge lists are not reclaimed or deallocated when all the edges of a vertex have been removed, and thus there are not any calls to the memory management. This allows for more efficient processing and consistent performance for edge deletions (in comparison with edge insertions). This is depicted in Fig. 2 where the deletion is consistent across for all graphs given a specific batch and batch size. Notice that there is very little variance across these experiments. In fact, the error bars are barely visible for deletions, whilst they are visible for edge insertions. Lastly, given large enough batch sizes, greater than 10k updates per batch, cuSTINGER supports almost 20 million edge deletions per second.

C. Triangle Counting

In this section, we take the popular triangle counting metric, which is part of clustering coefficients [28], [26], and check its performance using cuSTINGER. We consider the GPU algorithm from [13] which counts triangles in static graphs and port the implementation to cuSTINGER. While this is a static graph algorithm and not a dynamic graph algorithm for which cuSTINGER was designed, we show that cuSTINGER also offers good performance for static graph algorithms.

Table I has the execution time for both the original implementation (denoted as TC-CSR) as well as the new cuSTINGER implementation (denoted as TC-cuS). Both

algorithms output the same and correct number of triangles. From a performance perspective, there is a slight reduction in performance between these two implementation, where cuSTINGER is 1%-10% slower. This supports our intuition that our new data structure can also replace other static graph representations for the GPU.

V. Conclusions

In this paper we present cuSTINGER, the first dynamic graph data structure for the GPU. cuSTINGER manages the graph data structure on the GPU, allowing programmers to focus on algorithm development rather than complex data management and movement. cuSTINGER supports several different meta-data modes with run-time parameters. These modes allow controlling the amount of allocated memory on the GPU device, including restricting the amount of memory to be similar to that required by a CSR representation or full semantic support (as found in STINGER). Given the limited amount of storage available on the GPU this is crucial.

cuSTINGER supports extremely high update rates, from 15k updates per second with the graph being updated one edge at a time and well over 10 million updates per second when the batches are large (with over 100k updates per batch). And while cuSTINGER was designed with dynamic graph algorithms in mind, it can also be used for static graph algorithms. To benchmark cuSTINGER performance in comparison with CSR, we take a triangle counting algorithm originally implemented in CSR and port it to cuSTINGER. We showed that cuSTINGER performs within 10% of the original implementation.

Lastly, our goal is to improve on the current restrictions and overheads imposed by memory allocation. For future work on cuSTINGER we will look for an alternative approach for memory management for larger graphs and improved memory reclamation.

Acknowledgments

The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited." This work was also partially sponsored by NSF Grant ACI-1339745 (XScala).

References

[1] "NVIDIA CUDA Programming Guide," 2011.

[2] *Stanford Network Analysis Package*, 2012 (accessed April 2012). [Online]. Available: http://snap.stanford.edu/data/

[3] D. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. Poulos, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation," Georgia Institute of Technology, Tech. Rep., 2009.

[4] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering. 10th DIMACS Implementation Challenge Workshop*, ser. Contemporary Mathematics, no. 588, 2013.

[5] BlazeGraph, "https://www.blazegraph.com/," 2015.

[6] S. Che, "GASCL: A vertex-centric graph model for GPUs," in *IEEE High Performance Embedded Computing Workshop (HPEC)*, 2014.

[7] S. Che, B. M. Beckmann, and S. K. Reinhardt, "Belred: Constructing GPGPU graph applications with software building blocks," in *IEEE High Performance Embedded Computing (HPEC)*, 2014.

[8] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 349–359.

[9] D. Ediger., K. Jiang, J. Riedy, and D. Bader, "Massive Streaming Data Analytics: A Case Study with Clustering Coefficients," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[10] D. Ediger, R. McColl, J. Riedy, and D. Bader, "STINGER: High Performance Data Structure for Streaming Graphs," in *IEEE High Performance Embedded Computing Workshop (HPEC 2012)*, Waltham, MA, 2012, pp. 1–5.

[11] G. Feng, X. Meng, and K. Ammar, "DISTINGER: A distributed graph data structure for massive dynamic graph processing," in *IEEE Int'l Conf. on Big Data (Big Data)*, 2015, pp. 1814–1822.

[12] O. Green, R. McColl, and D. Bader, "A Fast Algorithm For Streaming Betweenness Centrality," in *4th ASE/IEEE International Conference on Social Computing (SocialCom)*, 2012.

[13] O. Green, P. Yalamanchili, and L. Munguía, "Fast triangle counting on the GPU," in *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[14] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: Efficient graph analytics using large multiversioned arrays," in *31st IEEE Int'l Conf. on Data Engineering (ICDE)*, 2015, pp. 363–374.

[15] R. McColl, D. Ediger, J. Poovey, D. Campbell, and D. Bader, "A performance evaluation of open source graph databases," in *ACM Workshop on Parallel Programming for Analytics Applications (PPAA)*, 2014, pp. 11–18.

[16] R. McColl, O. Green, and D. Bader., "Parallel Streaming Connected Components Using Parent-Neighbor Subgraphs," in *IEEE International Conference on High Performance Computing*, 2013.

[17] A. McLaughlin and D. Bader, "Revisiting edge and node parallelism for dynamic GPU graph analytics," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2014, pp. 1396–1406.

[18] ——, "Scalable and high performance betweenness centrality on the GPU," in *ACM/IEEE Conference on Supercomputing*, 2014, pp. 572–583.

[19] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 117–128.

[20] NVIDIA, "nvGraph," 2016.

[21] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-GPU graph analytics," *arXiv preprint arXiv:1504.04804*, 2015.

[22] A. Polak, "Counting triangles in large graphs on GPU," *arXiv preprint arXiv:1503.00576*, 2015.

[23] A. E. Sariyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek, "Betweenness Centrality on GPUs and Heterogeneous Architectures," in *6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013, pp. 76–85.

[24] J. Soman, K. Kishore, and P. Narayanan, "A fast GPU algorithm for graph connectivity," 2010.

[25] J. Soman and A. Narang, "Fast community detection algorithm with GPUs and multicore architectures," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2011, pp. 568–579.

[26] T. Schank and D. Wagner, "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study," in *Experimental & Efficient Algorithms*. Springer, 2005, pp. 606–609.

[27] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 265–266.

[28] D. J. Watts and S. H. Strogatz, "Collective Dynamics of 'Small-World' Networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.