# Autonomous, Independent Management of Dynamic Graphs on GPUs

Martin Winter
*Graz University of Technology*
Graz, Austria
martin.winter@icg.tugraz.at

Rhaleb Zayer
*Max Planck Institute for Informatics*
*Saarland Informatics Campus*
Saarbrücken, Germany
rzayer@mpi-inf.mpg.de

Markus Steinberger
*Graz University of Technology*
Graz, Austria
steinberger@icg.tugraz.at

***Abstract*—In this paper, we present a new, dynamic graph data structure, built to deliver high update rates while keeping a low memory footprint using autonomous memory management directly on the GPU. By transferring the memory management to the GPU, efficient updating of the graph structure and fast initialization times are enabled as no additional memory allocation calls or reallocation procedures are necessary since they are handled directly on the device. In comparison to previous work, this optimized approach allows for significantly lower initialization times (up to 300x faster) and much higher update rates for significant changes to the graph structure and equal rates for small changes. The framework provides different update implementations tailored specifically to different graph properties, enabling over 100 million of updates per second and keeping tens of millions of vertices and hundreds of millions of edges in memory without transferring data back and forth between device and host.**

## I. Introduction

In today's world, large, ever-changing data structures are in common use and dynamic graphs are used in many application domains, ranging from communication networks, to social media networks, to intelligence data. And as graphics processing units (GPUs) become ever more ubiquitous and comparatively inexpensive, massively parallel compute devices are available to deal with problems posed in such large-scale domains. Currently, there are many static graph libraries and algorithms available that take advantage of this massive parallelism. However, most deal only with the static use case, as performance on the GPU is predicated on being able to managed thread divergence, memory locality and optimal work distribution, which becomes increasingly difficult in a dynamic setting, where the memory layout is constantly changing.

In this paper, we present an agile and fast dynamic graph solution on the GPU with low memory requirements and autonomous memory management. Our implementation uses CUDA [1], but could also be implemented using OpenCL [2]. The proposed solution is built from the ground up to achieve maximum performance on the GPU, allocating the free device memory upfront and managing all memory directly on the GPU. This alleviates individual reallocation calls for dynamic graph updates and allows the framework to perform edge insertions and deletions with a single kernel call respectively. The host only provides update data to the framework, which performs the update independently on the GPU.

An added benefit of this methodology, in addition to alleviating additional management interventions from the host, is the fact that users do not need to care about memory management themselves. The system provides not only a mechanism to hold and update the graph data structure on the GPU, but also the ability to place temporary data (e.g. edge updates, algorithmic data) on the device without the need to allocate or free this memory segment later on.

The general memory layout is tailored to the architecture of the GPU, providing a management segment at the beginning of the device memory that allows to perform locking on a per-vertex basis if so required. The adjacency per vertex itself is stored in a combination of an edge block array and an edge block list, allowing for efficient memory access within a block but keeping the flexibility to resize the adjacency with little overhead during reallocation. The framework provides three different semantic modes for graphs, allowing to include weight, type and timestamp information, which provide additional expressiveness at the cost of higher memory requirements.

Depending on graph properties (like the average degree per vertex), different update implementations are provided that are optimized for the different properties. If the average size of an adjacency per vertex is small, a different update algorithm performs better when compared to large adjacencies. These optimizations are turned on by default, but can also be adjusted by the user depending on the use case.

## II. Related Work

### A. Static Graph Algorithms on the GPU

There are a variety of static graph data structures available on the GPU, ranging from Gunrock [3] to nvGraph [4], BlazeGraph [5], GasCL [6] and BelRed [7]. nvGraph [4] offers implementations of *page rank*, *single source shortest path* as well as *single source widest path* on the GPU and is freely available as part of the *CUDA toolkit*. Gunrock [3] is a CUDA library for graph processing using highly optimized operators for graph traversal while achieving a balance between performance and applicability to a wide range of problems. BlazeGraph [5] offers a high-performance graph database, using its own domain-specific language, DASL, to implement advanced analytics with high-level functionality.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

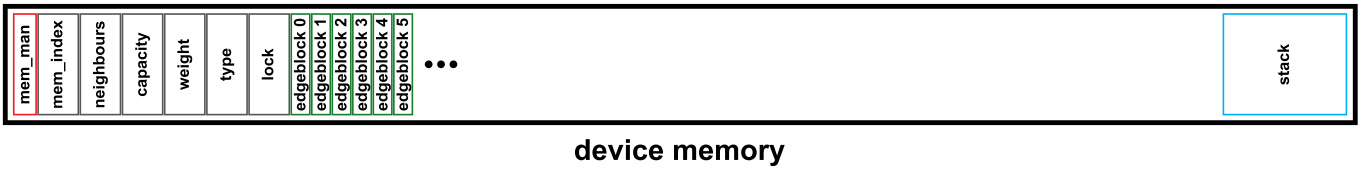


Fig. 1. Visualization of the device memory layout as managed by the *memory manager*

### B. STINGER

Spatio-Temporal Interaction Networks and Graphs Extensible Representation (**STINGER**) [8] is a high performance data structure designed for efficient handling of dynamic graphs. It uses a blocked linked list approach to store the adjacency data per vertex. In its advancement, **GTSTINGER** [9], an internal memory manager is used for allocating such edge blocks to the individual vertices. **GTSTINGER** shows that high update rates are possible, outperforming several leading graph databases, including shared and distributed-memory based approaches.

### C. cuSTINGER

**cuSTINGER** [10] is a new graph data structure focussing on streaming graphs and represents a GPU implementation of **STINGER**. They provide different meta-data modes for edges and are capable of high update rates by addressing **cuSTINGER** addressess the key issues of the **STINGER** data structure on the GPU, switching from edge block lists to edge block arrays, interleaving the vertex management data and allocating individual edge block arrays as a whole. However, the restrictions of the **STINGER** data structure do not fit well on the GPU, as the management data structures are unnecessarily large and GPU utilization depends on the average size of the adjacency per vertex. For small to medium sized adjacencies (less than 50 vertices per adjacency), this leads to a unnecessarily large number of stalling threads per update. Initializing the data structure is performed from the host and updates hit a bottleneck once reallocations are necessary. **cuSTINGER** tries to deal with this issue by over-allocating 50% more memory per adjacency, which helps in the beginning or for small update batches, but hampers performance especially in continuous mode or for large update batches Additionally, this places tighter restrictions on the size of graphs, as this over-allocation becomes significant for larger graphs.

## III. aimGraph

In the following section we sketch the design and general idea behind **aimGraph** (**a**utonomous, **i**ndependent **m**anagement of dynamic **Graph**s on the GPU) and focus on the initial memory setup and layout, the update implementations as well as performance relevant optimizations. This section is followed by a comparison to **cuSTINGER** and a look at performance differences.

### A. Memory Layout

**aimGraph** initializes the system with a single GPU memory allocation, assigning as much memory as is available on the device to the framework. All following allocation calls are handled internally by requesting memory from a simple memory manager. This simple memory manager is initialized from the CPU (setting up the edge mode, the block size, kernel launch parameters) and then placed at the beginning of the large block of memory previously allocated on the device. It holds a pointer to the beginning and end of the allocated device memory and also stores all the necessary management data. Using this autonomous memory management approach, the framework can facilitate all dynamic memory needs directly on the GPU and significantly reduce the run time overhead by avoiding individual allocation calls from the host.

Our memory manager follows a similar approach as traditional memory management in CPU C/C++ program. Static data is placed at the bottom, the dynamic heap area is placed right after the static data, while the temporary data on the stack grows from the top. We consider these three regions in our memory manager on the GPU.

*1) Static data:* Similarly to **cuSTINGER**, the number of vertices is considered static in our current implementation. Adding or deleting vertices from the graph is not supported, hence the size of the static data segment is known at the time of initialization. As previously mentioned, right at the start of the application, the **memory manager** and after that vertex management data structures are placed in device memory. The management data is set up as a structure of arrays, each array the size of $numberVertices \cdot sizeof(parameter)$. The parameters in question are

- **memindex**: Holds the block index of a dynamic edge block, where the adjacency data starts
- **neighbours**: Holds the number of neighbours in the adjacency
- **capacity**: Holds the maximum number of neighbours in the adjacency with the current block allocation
- **weight**: A weight can be assigned to each vertex, optional parameter
- **type**: A type can be assigned to each vertex, optional parameter
- **lock**: Algorithms can restrict access to individual vertices using this lock

All individual arrays are placed memory aligned, each at least the size of a multiple of the general GPU cacheline size of *128 bytes*, each vertex hence requires at least $6 \cdot 4$ bytes $= 24$ bytes.

*2) Dynamic data:* The data after the static data segment is managed in blocks, the block size depends on the application and the size of the edge data. Each block stores adjacency data and uses the last *4 bytes* to indicate the location of the following block. For a simple adjacency storing just the destination vertex, *64 bytes* suffices, for semantic graphs, the block size is larger to accommodate more vertices per block. Additionally, different update mechanisms profit from different memory block sizes,

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

depending on the update strategy and the average size of the adjacency per vertex an optimal block size for the given graph is chosen.

In the initialization step, even this data can be set up with maximum parallelism using an *exclusive prefix scan* to determine the memory requirements for each adjacency list in a pre-computation step. The last element in an edge block is always an index to the next edge block. This makes this approach a combination of a linked list and an adjacency array and allows for memory locality for vertices within an array. At the same time, this strategy avoids reallocation of the whole block if augmentation is required, as another block can be allocated by simply updating the index at the end of the last block. This approach also leaves the possibility to switch to more sophisticated memory management in the future.

*3) Temporary data:* In the initialization phase, but also for updating the graph and algorithms running on the graph, additional, temporary data may be required. This can, e.g., be edge updates (consisting of source and destination vertex data) or an array holding the triangle count per vertex to calculate the overall triangle count within a graph structure.

This data is managed like a stack. The memory manager holds a stack pointer pointing to the end of the allocated memory and can deal out shares of this memory to algorithms or for pushing updates to the graph structure. This way the whole device memory can be managed without considering a trade-off between the managed memory portion of the device memory and the temporary data needed, the memory manager just has to check if temporary data does not protrude into the dynamic data segment.

### B. Initialization

At the start of the application, a graph is parsed into an intermediary CSR (Compressed Sparse Row) format and in the beginning, a preprocessing kernel is started to calculate the memory requirements per vertex, in detail computing the number of `neighbours` and from that the `capacity` and `block requirements` per vertex in parallel. Using the `block requirements` and an *exclusive prefix sum scan*, the overall memory offsets for all individual edge block lists can be computed.

Using all this information, the initialization kernel can be run completely in parallel without regard for locking, while the CSR format is transferred into the **aimGraph** format, only a single instruction is performed by a single (the last) thread, as this one can set the `next_free_block_index` in the memory manager, which is required for dynamic memory allocation for edge updates.

### C. Edge Types

**aimGraph** supports three different edge types:

- **Simple**: This mode stores the bare graph structure using just the destination vertex in the edge data array
- **Weights**: This mode adds the support for weights for both vertices and edges to the simple mode
- **Semantic**: Additionally to weights, this mode adds support for type information for both vertices and edges and also two timestamps per edge, which increases the size required per edge significantly

This variety of options is implemented using templated classes and methods, as most functionality is independent of the concrete representation of the edges themselves, just the modification functionality is realized via overloaded functions. Depending on the use case, one of these more advanced modes can be selected at the cost of an increased memory footprint, choosing a larger sized edge type also increases the basic block size to accommodate a larger number of edges per block.

**Algorithm 1:** Edge insertion using locking

```
Data: edge update batch
Result: Edges inserted into graph
Edge updates put onto stack;
while lock acquired do
    read neighbours & capacity;
    for vertices v in adjacency do
        if v == DELETIONMARKER || index ≥
         neighbours then
            remember index;
            break;
        if v == edge update then
            found duplicate, ignore;
            break;
        advance in EdgeBlockList;
    if !edgeInserted && !duplicateFound then
        get memBlock from memManager;
        update adjacency, index, capacity & neighbours;
    else if !duplicateFound then
        insert element at index;
    release lock;
```

### D. Edge Insertion

Edge updates in the current setup require a single lock per vertex to combat concurrent read/writes to the adjacency, neighbours and capacity as shown in algorithm 1. Access to the *memory manager* on the other hand simply requires *atomic memory access* to get a new block, if the current capacity cannot accommodate the new update.

This results in a high update rate, if the edge updates do not particularly favor a small set of vertices over the majority. For edge updates that are close to a uniform random distribution, inserting $1.000.000$ edges can be achieved in a few milliseconds. Depending on the average size of the adjacency, even when accessing the memory manager heavily to adjust the size of individual edge block lists.

We provide two implementations, optimized for different adjacency list counts. The first, which is the standard insertion mode, is shown in listing 1. It completes each individual update

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

using a single thread. This approach is especially fast for small to medium sized adjacency lists (less than 50 vertices).

If the average size per adjacency grows larger, the traversal of the graph structure becomes the bottleneck. Thus, for larger adjacency list sizes, we use an entire warp (32 threads) for the update. In this case the *for-Loop* reduces to a loop over blocks and the memory access pattern within blocks can be optimized to requesting a full cacheline per warp at once.

### E. Edge Deletion

Edge Deletion works in a similar manner to edge insertion, the major difference results in the fact that there is no need to access the memory manager, as no new memory will be required. Additionally, we do not return empty blocks to the memory manager, but simply reuse them when edges are inserted for the same node again. In this way, we can avoid access to the memory manager completely during deletion.

As with the insertion process, two different implementations are provided, one launching a single thread per update and the other launching a full warp per update, depending on the average size of the adjacency. In the following, we show the deletion process for the standard launch. When launching a full warp per update, the *for-Loop* is again reduced and leads to better memory locality as a whole cacheline is fetched by the warp.

**Algorithm 2:** Edge deletion without locking

```
Data: edge update batch
Result: Edges deleted from graph
Edge updates put onto stack;
read capacity;
for vertices v in adjacency do
    if v == edge update then
        atomically update Adjacency & neighbours;
        one thread decreases neighbours;
        break;
    advance in EdgeBlockList;
```

## IV. Comparison to cuSTINGER

This section provides a comparison between **aimGraph** and **cuSTINGER** by looking at the respective memory footprints and composition, as well as the time spent initializing and updating the graph structure and is followed by an evaluation of the performance differences.

### A. Memory footprint

One of the biggest differences stems from the way memory allocation is performed in general, **cuSTINGER** performs individual calls to *cudaMalloc()* from the CPU to allocate the management data and all individual edge blocks. Especially for graphs with more than a million vertices this is a significant overhead, compared to the single allocation in **aimGraph**.

Another big difference lies in the memory footprint. **cuSTINGER** uses pointers to (a) locate attributes, (b) to point to individual edge blocks, (c) to point to data members within an edge block (especially prevalent in *semantic* mode). This increases the size of the management data set and also requires a full block (of *64 bytes*) just to hold member pointers.

**aimGraph** on the other hand uses an indexing system (reducing the size per pointer/index from *8 bytes* to *4 bytes*), but also eliminates the member pointers and additional attribute pointers by combining an efficient indexing scheme and reinterpreting memory on the fly using casts to achieve the same functionality at a fraction of the memory.

### B. Initialization

As previously mentioned, **aimGraph** performs a single device memory allocation and can perform the whole setup in parallel on the GPU with little overhead. Compared to that, **cuSTINGER** needs to allocate each individual edge block list from the CPU, also performing the pre-computation entirely on the CPU and only the actual setup of the data structure occurs on the GPU. However, as there is no offset-indexing scheme, even this launch cannot utilize the GPU to its full potential, leading to an enormous performance difference in the initialization stage.

### C. Updates

Here once again, the different strategy in allocating memory pays off for **aimGraph**, as updates can be achieved in a single kernel launch with a single lock per vertex when inserting edges and even without a lock in the deletion process.

**cuSTINGER**, on the other hand, launches at least one kernel, which cannot utilize the whole GPU due to the lack of locking, but also incurs a heavy penalty if duplicates are present or reallocation is required. In this case, new space must be allocated using *cudaMalloc()* and the whole edge block list of the given vertex needs to copied over. In the worst case 5 kernel launches are required to deal with all eventualities, leading to significantly lower update rates.

**cuSTINGER** holds a slight edge in case no duplicates are in the batch, no reallocation is necessary and the average size of an adjacency is large (greater than 50), as only few operations are performed. However, in these cases there is a chance to produce invalid graphs, as there is no locking or contention resolution mechanism in place. Depending on the actual behavior of the hardware thread scheduler this problem may show up more or less often.

## V. Performance

The performance measurements were conducted using a NVIDIA GTX 780 GPU (3 GB V-RAM), an Intel Core i7-3770K using 16 GB of DDR3-1600 RAM. The GTX780 is a Kepler based card with CC 3.5, and equipped with 2496 CUDA Cores, 12 SMs and 192 TMUs per SM.

Although this is considered consumer hardware, the goal is to show differences between **aimGraph** and **cuSTINGER**. Performance on more powerful professional equipment is expected to be even higher. The graphs used were taken from the 10th DIMACS Graph Implementation Challenge [11] and

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

a selection used for performance testing is highlighted in table I. Both frameworks use the same testing methodology, starting with initialization, followed by the generation of random edge updates, which were subsequently added to the graph and then removed again. This is done 10 times and the results are averaged to produce the overall results. Only the calls to the initialization and update functions were measured. This whole process is repeated 10 times and averaged again, hence the performance numbers shown display the average time of 10 rounds of initialization and 100 rounds of edge insertions and deletions respectively.

| Name | Network Type | \|V\| | \|E\| |
|---|---|---|---|
| Luxembourg | Road | 115k | 239k |
| coAuthorsDBLP | Citation | 299k | 1.95M |
| ldoor | Matrix | 952k | 45.57M |
| audikw1 | Matrix | 943k | 76.71M |
| Germany | Road | 12M | 24.74M |
| nlpkkt160 | Matrix | 8M | 221.17M |

TABLE I
GRAPHS USED FOR PERFORMANCE MEASUREMENT

### A. Initialization

As shown in table II, the different memory setup procedure pays off the most in the initialization step, the highest advantage is achieved when processing a high number of vertices with a comparatively low number of edges. In this case, **aimGraph** is nearly 300 times faster. Even for a low number of vertices with a high degree the speed up achieved still reaches double digits. This can be attributed to the fact, that **aimGraph** works autonomously on the GPU and can parallelize the setup process. In contrast, **cuSTINGER** performs its setup process from the host with individual initialization calls per vertex, calculating memory requirements and allocating memory from the host directly.

Additionally, as **aimGraph** has a significantly lower memory footprint. Thus, larger graphs can be kept in memory compared to **cuSTINGER** as can be seen for the sparse Matrix network `nlpkkt160`.

| Name | Initialization (ms) aimGraph | Initialization (ms) cuSTINGER |
|---|---|---|
| Luxembourg | 3.13 | 110.5 |
| coAuthorsDBLP | 6.687 | 289.6 |
| ldoor | 53.704 | 1 053.2 |
| audikw1 | 86.713 | 1 108.6 |
| Germany | 101.68 | 14 010.7 |
| nlpkkt160 | 228.13 | out of memory |

TABLE II
INITIALIZATION TIME IN *ms* FOR **AIMGRAPH** AND **CUSTINGER**

### B. Edge insertion

The first three cases in figure 2 show where **aimGraph** has a clear performance advantage. This is the case if the degree per vertex is small, as in those cases the over-allocation strategy of

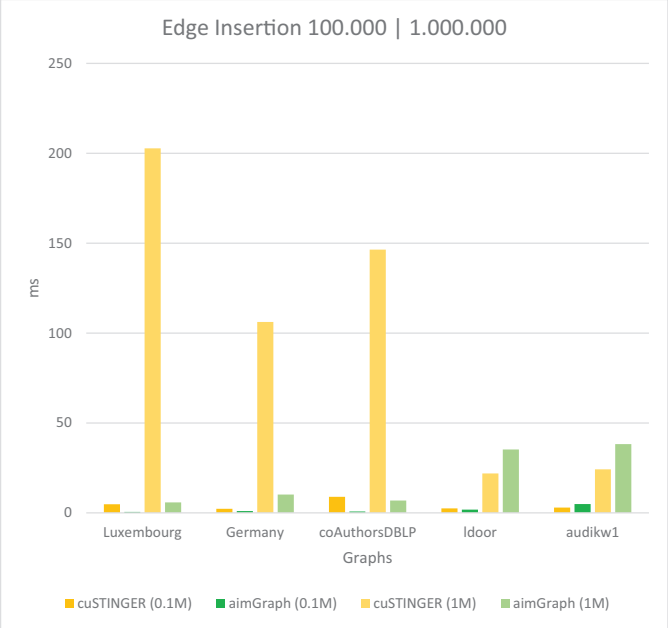


Fig. 2. Performance measurement for edge insertions, using a batch size of 100.000 and 1.000.000

**cuSTINGER** does not provide enough space for the insertion operations and both frameworks have to reallocate which is much faster using **aimGraph**, as everything is done on the GPU in one kernel. **cuSTINGER** has to reallocate from the CPU and also copy over entire edge blocks.

**cuSTINGER** has an advantage due to their over allocation policy. **cuSTINGER** allocates 50% more to reduce the need for reallocation later on, if there is a comparatively low number of vertices compared to the number of edges (as seen with the sparse matrices). In these cases, **cuSTINGER** achieves the updates in less time than **aimGraph**, as we actually have to perform memory allocations, which involve more complex traversal mechanisms and locking.

Additionally, as **cuSTINGER** does not use any form of race condition avoidance, there might arise some cases which result in an invalid graph structure. Depending on GPU scheduling, duplicates within batches are not detected and remain in the graph. The behavior of **aimGraph** is independent of scheduling, and keeps a more compact memory layout. Although, we employ correctness guaranties and keep memory requirements significantly lower, **cuSTINGER** only shows a slight performance advantage.

Another factor, which becomes performance relevant, is the difference in adjacency traversal. Due to the more modular structure of **aimGraph**, the traversal of individual edge lists takes longer compared to the array traversal of **cuSTINGER**, as the indexing scheme behind connecting multiple blocks into a contiguous list requires extra cycles.

For testing purposes, turning off/down the memory over-allocation of **cuSTINGER** decreases performance up to a factor of 100 or changing the update strategy by first inserting 10 batches of updates and then removing them again also worsens

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

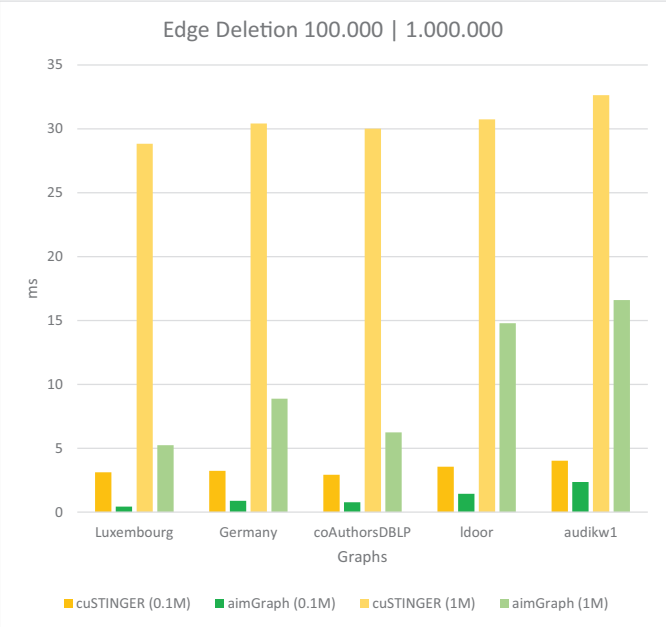


Fig. 3. Performance measurement for edge deletions, using batch size 100.000 and 1.000.000

performance for **cuSTINGER** significantly, while **aimGraph** does not see a major effect on its performance. Overall, it can be noted that cuSTINGER only performs well when it works within its overallocation boundaries and for larger sized adjacencies. Otherwise its performance significantly drops.

### C. Edge Deletion

In case of deletions, the performance difference is slightly less pronounced compared to the insertion process as can be seen in figure 3. This is due to the fact that deletions always work without rearranging the general memory layout and also there exists no possibility of adding/removing duplicates.

**aimGraph** uses variant 1 of the deletion procedures for the first four graphs (one thread per update), as the average adjacency is comparatively small to medium sized (less than 50 vertices per vertex on average). Under these circumstances adjacency traversal is less important compared to stalling threads. The performance benefit is therefore greatest for very small adjacencies per vertex and becomes less prominent for larger adjacencies.

The last case uses variant 2, launching warp-sized blocks, as in those cases the adjacency traversal is crucial to performance, and once again performance is about 2× faster compared to **cuSTINGER** for the tested graphs. The main difference to **cuSTINGER** is the single kernel launch (compared to two launches for **cuSTINGER**) and the more efficient duplicate checking.

## VI. Conclusion

**aimGraph** is a memory-efficient streaming graph solution on the GPU that enables very high update rates without the need to transfer the graph data structure to and from the host. This solution is purpose-built for the GPU, keeping memory requirements low by using an indexing structure instead of pointers and managing the device memory on the device autonomously, without the need for copying and allocating new blocks from the host. In this way, updating the graph structure can be achieved with a single kernel call respectively and allows for concurrent initialization and updates.

The current implementation includes support for different semantic modes (including simple, weighted and semantic graphs) and offers developers different updating strategies, which can be selected for specific workloads for optimal performance. Also different verification methods are present to test and verify new features and algorithms. Even on consumer-level GPUs (NVIDIA GTX 780 with 3GB VRAM), the framework can hold tens of millions of vertices and hundreds of millions of edges in memory (depending on the semantic mode) and is also able to process 20 - 100 million insertions per second and between 50 - 150 million deletions per second.

Overall, **aimGraph** offers an efficient and fast dynamic graph implementation with low memory footprint and autonomous memory management, allowing for different update mechanisms tailored to different graph properties.

## VII. Future Work

In its current form, **aimGraph** offers a streaming graph solution, capable of high update rates while keeping the memory requirements as low as possible. To further update and expand the capabilities, the objective is to assess more advanced memory management techniques. This could be achieved by a queueing approach to have a more modular solution and also save space, especially when using the framework in continuous mode, the current indexing structure and mixed approach should translate well to this more advanced approach.

Furthermore, we would like to investigate the use of a mega kernel approach, launching just a single kernel and distributing the resources on the fly, eliminating separate kernel launches from the host all together. This would also incorporate the possibility of updating the graph structure and running algorithms simultaneously.

Currently, the testing procedure focusses mainly on initializing and updating the graph structure and deriving performance metrics from this data, in the future the objective would be to implement different graph algorithms, including triangle counting [12], [13], connected components [14], single-source shortest path [15], betweeness centrality for static graphs [16], betweeness centrality for dynamics graphs [17] and community detection [18].

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

## ACKNOWLEDGMENTS

This research was supported by the DFG grant STE 2565/1-1, the Austrian Science Fund (FWF) I 3007 and the Max Planck Center for Visual Computing and Communication.

## REFERENCES

[1] "NVIDIA CUDA programming guide," 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[2] "Opencl - the open standard for parallel programming of heterogeneous systems," 2017. [Online]. Available: https://www.khronos.org/opencl/

[3] 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*, 2015.

[4] "nvgraph," 2016. [Online]. Available: https://developer.nvidia.com/nvgraph

[5] "Blazegraph," 2017. [Online]. Available: https://www.blazegraph.com/

[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] D. Bader, J. Berry, A. Amos-Binks, D. Chavarria-Miranda, C. Hastings, K. Madduri, and S. Poulos, "STINGER: Spatio-temporal interaction networks and graphs (STING) extensible representation," in *Tech. Rep.* Georgia Institute of Technology, 2009.

[9] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *IEEE High Performance Extreme Computing Conference (HPEC)*. Georgia Institute of Technology, 2012.

[10] O. Green and D. Bader, "cuSTINGER: Supporting dynamic graph algorithms for GPUs," in *IEEE High Performance Extreme Computing Conference (HPEC)*. Georgia Institute of Technology, 2016.

[11] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering. 10th dimacs implementation challenge workshop," in *ser. Contemporary Mathematics, no. 588*, 2013.

[12] O. Green, P. Yalamanchili, and L. Munguia, "Fast triangle counting on the GPU," in *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, 2014.

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

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

[15] 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.

[16] A. E. Sariyüce, K. Kaya, E. Saule, and . V. Catalyürek, "Betweenness centrality on GPUs and heterogeneous architectures," in *6th Workshop on General Purpose Processor Using Graphis Processing Units*, 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.

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

978-1-5386-3472-1/17/$31.00 ©2017 IEEE