faimGraph: High Performance Management of Fully-Dynamic Graphs Under Tight Memory Constraints on the GPU

Martin Winter*, Daniel Mlakar*, Rhaleb Zayer†, Hans-Peter Seidel† and Markus Steinberger*†
*Graz University of Technology, Austria
Email: {winter, mlakar, steinberger}@icg.tugraz.at
†Max Planck Institute for Informatics, Germany
Email: {rzayer, hpseidel}@mpi-inf.mpg.de

***Abstract*—In this paper, we present a fully-dynamic graph data structure for the Graphics Processing Unit (GPU). It delivers high update rates while keeping a low memory footprint using autonomous memory management directly on the GPU. The data structure is fully-dynamic, allowing not only for edge but also vertex updates. Performing the memory management on the GPU allows for fast initialization times and efficient update procedures without additional intervention or reallocation procedures from the host. Our optimized approach performs initialization completely in parallel; up to 300x faster compared to previous work. It achieves up to 200 million edge updates per second for sorted and unsorted update batches; up to 30x faster than previous work. Furthermore, it can perform more than 300 million adjacency queries and millions of vertex updates per second. On account of efficient memory management techniques like a queuing approach, currently unused memory is reused later on by the framework, permitting the storage of tens of millions of vertices and hundreds of millions of edges in GPU memory. We evaluate algorithmic performance using a PageRank and a Static Triangle Counting (STC) implementation, demonstrating the suitability of the framework even for memory access intensive algorithms.**

***Index Terms*—Parallel programming, GPU, Massively parallel algorithms, Dynamic graphs**

I. Introduction

Dynamic graphs are commonly used to model and analyze the large ever-changing data arising across multiple fields. This includes *communication networks*, where vertices model mobile devices with the connections between them or cell towers represented by edges; *social-media networks*, where vertices may represent people with edges indicating friend relationships; and *intelligence networks*, where vertices model agents with edges highlighting their interactions. In combination with the rise of *big data*, there is an immanent need for highly efficient, dynamic graph data structures that support millions of vertices and edges, which can change constantly.

As the modern Graphics Processing Unit (*GPU*) becomes ever more ubiquitous and comparatively inexpensive, the GPU seems predestined to deal with this large-scale problem domain. Leaving aside the cost factor, the turn to massively parallel architectures like the GPU is also justified by limitations of hardware manufacturing: In the past, it was expected that every new hardware generation increases its clock speed and transistor count. Although the transistor count keeps growing exponentially, the clock speed has hit the so-called *power wall* [1]. Consequently, a significant speed-up is only possible by exploiting parallelism. This trend towards rapid increases of core counts can especially be observed on the GPU, where new generations increase core counts by multiple thousand.

In general, the GPU has gained a lot of traction as a general purpose processor. Its Single Instruction - Multiple Data (SIMD) processing model lends itself to problems with high data parallelism, which is typical also in graphs, especially as they increase in size. Furthermore, the throughput-oriented architecture of the GPU also fits the graph domain well. Since most graph algorithms are comprised of simple operations that have to be performed on millions of objects, the GPU seems like an ideal candidate. Thus, it is not surprising that various *static graph libraries* target the GPU [2]–[6].

However, dynamic memory management for a large number of entities—as required for *dynamic graphs*—is challenging on the GPU. The GPU performs best, when memory requirements and layout are known beforehand to allow appropriate optimizations. Especially changing memory requirements are difficult to handle. Typically, memory allocation on the GPU is handled by the Central Processing Unit (CPU), which disrupts parallel execution on the GPU. While efficient memory allocation already forms an issue for dynamic GPU execution, freeing memory is an even bigger challenge: As many small memory deallocations yield large overheads, freed memory is often simply not returned to the system. Over time, such strategies lead to memory fragmentation, reduce the available memory, and ultimately lead to system failure as it runs out of memory.

Furthermore, unbalanced graphs lead to an unbalanced work loads. As the structure of dynamic graphs continuously changes, load balancing strategies also need to adapt to achieve high performance. Thus, performing load balancing with performance influencing factors in mind, like thread divergence on the SIMD cores of the GPU and memory locality, becomes an even more challenging task for dynamic graphs. Probably due to these issues, the number of dynamic graph frameworks for the GPU is very limited, namely, *aimGraph* [7],

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

cuSTINGER [8], *GPMA* [9] and the upcoming Hornet [10]. None of these support fully dynamic graphs, but consider the graph's vertices fixed and only support dynamic edge data and/or updating individual properties of existing vertices. Furthermore, their memory management strategies worsen over time, leading to lavish use of memory or even system failure.

With our approach, faimGraph, we tackle these issues, presenting a *fully dynamic* GPU framework for large dynamic graphs, which is completely GPU-autonomous, reclaims and reuses all freed memory, and reduces fragmentation to a minimum for both vertex and edge data. To achieve these goals, we introduce an advanced dynamic memory management system for the GPU tailored to graphs, which uses efficient queuing structures to reassign memory directly on the GPU in *O(1)*. Although more complex data management naturally increases memory access times, our framework achieves equal or better performance throughout both memory management routines and algorithms executing on top of the graph structure.

II. Related Work

Related work on graph data structures for the GPU can be categorized into static graph libraries, dynamic graph libraries, and GPU adapted implementations of graph algorithms.

A. Static Graph Frameworks on the GPU

There exists a variety of static graph data structures on the GPU: *nvGraph* [2] (NVIDIA Graph Analytics library) offers implementations of three widely-used algorithms, supporting up to two billion edges (using an NVIDIA Tesla M40 with 24 GB). *BlazeGraph* [3] offers a high-performance graph database, using its own domain-specific language, *DASL*, to implement advanced analytics. *BelRed* [4] addresses the problem that manual effort is often required to build graph application on the GPU. They introduce a library of software building blocks which can be combined to build graph applications. *Gunrock* [5] is a Compute Unified Device Architecture (CUDA) library for graph processing using highly optimized operators for graph traversal while achieving a balance between performance and applicability. *GasCL* [6], a vertex-centric graph model for the GPU, written in Open Computing Language (OpenCL), supports the "think-like-a-vertex" programming model.

While all these libraries achieve high performance for static graph algorithms, they do not consider dynamic changes of graphs. Thus, directly using any of these frameworks for dynamic graphs, would require a complete reallocation of the graph whenever any part of the graph changes. This is obviously not sustainable as graphs are changing often and therefore specialized solutions for dynamic graphs are required.

B. Dynamic Graph Frameworks on the GPU

At the time of writing, there exists only two published (apart from *aimGraph* [7]) frameworks for dynamic graphs on the GPU: *cuSTINGER* [8] and *GPMA* [9]. Furthermore, *Hornet* [10], the successor of *cuSTINGER*, is currently in development. *GPMA* [9] supports efficient stream updates on graphs and is suitable for high-speed analytical processing. The main feature of their approach is the implicit sorting of the adjacency data, which results from the use of an adapted Packed Memory Array (PMA) scheme. This is beneficial for algorithms that can exploit shortcuts on sorted adjacency data. Their downside is a significantly increased memory consumption compared to straight-forward arrays and the additional effort required during reallocation as the PMA is rebuilt.

cuSTINGER [8] is a GPU-adaptation of STINGER [11] and its internal memory manager [12]. Although *cuSTINGER* targets the GPU, many management tasks still require the CPU. Foremost, when a vertex adjacency has no space left, the CPU allocates new GPU memory and copies over the old data. To reduce the probability of costly CPU operations, *cuSTINGER* over-allocates all adjacency structures and does not free unused adjacencies. This leads to high memory fragmentation and memory waste, and over time potential system failure.

aimGraph [7] is similar to *cuSTINGER* as it also only supports static vertex data and can only grow adjacencies of each vertex. Its advantage is an autonomous memory management on the GPU, avoiding CPU round trips when adjacencies run out of memory. Due to the increased performance during allocation, it can get away with less overallocation while achieving similar or better performance than *cuSTINGER*.

In summary, the aforementioned libraries all require significant additional memory, potentially costly round-trips to the CPU, and may run out of memory over time. Furthermore, no framework supports dynamic vertices. Our work addresses all of these points, forming a first GPU framework for completely dynamic graphs that can run autonomously from the CPU and is able to hold much larger graphs, especially considering long-term use cases with highly volatile graph structures.

C. GPU Graph Algorithms

While a dynamic graph representation is certainly an important goal, algorithmic performance on top of the representation is as important. Among standard graph algorithms are graph metrics, many of which have been implemented on the GPU. These include triangle counting [13], [14], which can be used to find key players in a network based on their local connectivity; PageRank [15], which measures the importance of web pages according to the links to a page, connected components [16], single-source shortest path [17], the betweenness centrality [18], [19], and community detection [20]. To display the suitability of our framework for memory-intensive algorithms, we test the performance for PageRank and triangle counting.

III. faimGraph

In the following section we discuss the design of *faimGraph* (**f**ully-dynamic, **a**utonomous, **i**ndependent **m**anagement of **graph**s). We focus on the core contributions, which are a tight memory model building on the reuse of memory by utilizing queuing structures, dynamic changes of vertex and edge data, as well as high performance update implementations and algorithms running on top of the graph.

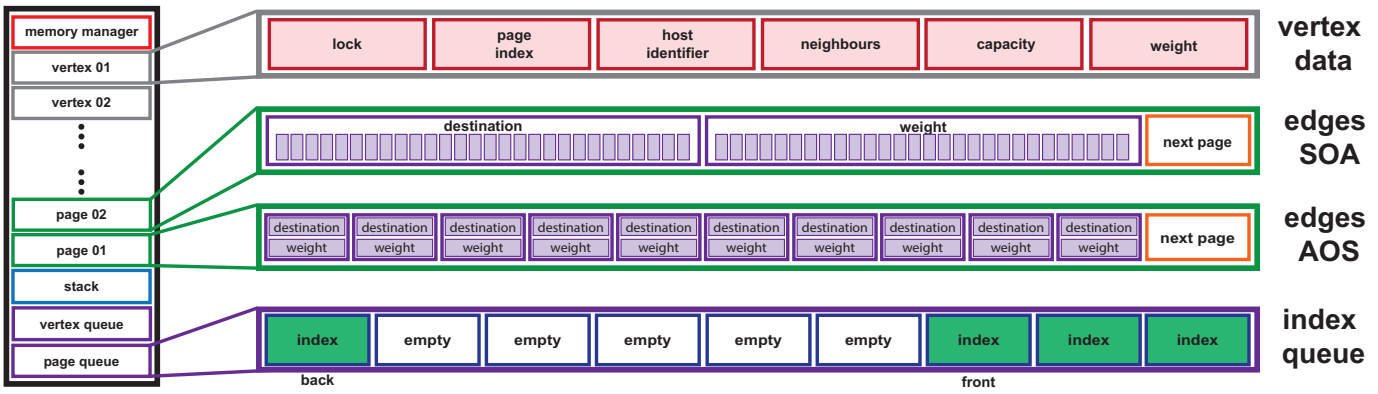

Fig. 1: Visualization of the device memory layout as managed by the *memory manager* for a **weighted** graph, displaying both available edge data layout options.

A. Memory Management

The central idea of *faimGraph* is performing all memory management directly on the GPU, requiring only a single allocation of a large block of memory to avoid round-trips to the CPU. This block serves all memory requirements for the graph structure itself as well for algorithms running on top of the graph. During initialization, *faimGraph* prepares this memory, as shown in Figure 1, to support dynamic assignment and reassignment using queuing structures to keep track of unused memory. A *memory manager* is used to keep track of individual memory sections and current graph properties, like number of vertices/edges, free pages, and unused vertex indices. The majority of the memory is used by dynamically allocated *vertex data* and pages for *edge data*. Both regions grow from opposite sides of the memory region, to not restrict the ratio between vertices and edges. Temporary data (updates or helper data structures) can be placed in a *stack* which is followed by two *queuing structures* for reclaiming freed vertices and edge pages. Since the complete addressing scheme uses relative indices, the framework can also be started with conservative memory bounds, as in most cases reinitialization is possible directly from old *faimGraph* to new *faimGraph* just building on just two `memcpy`'s on the device directly. If resources are even more scarce, reinitialization also can be performed (at a higher cost) from *device CSR*, *host CSR* or even *host* faimGraph.

B. Queues

The core entity for memory reclamation are the *index queues* used to store freed vertex indices and pages. Whenever a vertex is deleted or a page is freed, its index is pushed into the respective index queue. During resource allocation, threads at first attempt to pop a free element from the queue and only if that fails, increase the vertex or page region. Using this approach, changes in growth in the graph do not affect the required memory as much as previous approaches would have, as a graph can grow in specific areas and shrink back in others. Furthermore this allows for *O(1)* allocation of vertices as well as pages. For efficiency, we use array-based queues, which operate on top of a ring buffer of indices. The queues must support concurrent access from thousands of threads and efficient queries for empty states. Thus, we use a front and back pointer as well as a fill counter for each queue, similar to the broker queue [21]. Threads at first test the fill counter to determine whether there are elements in the queue. Only then, they atomically move the pointers to retrieve a queue element. As the entries in the queues are simple indices, we use atomic compare-and-swap (CAS) to insert or remove elements from the queue while using an empty flag to avoid read-before-write and write-before-read hazards.

C. Graph Data

1) Vertex Data: Other approaches, such as *aimGraph* [7] and *cuSTINGER* [8] consider vertex data as static. However, dynamic graphs may require the ability to add or remove vertices from the graph. While a static vertex management can follow a Structure of Arrays (SOA) approach to enable efficient memory access to this data on the GPU, such an approach interferes with the concept of dynamic data distribution between vertex and edge data (one would have to choose a fixed array size to place the arrays after each other in memory). Thus, we store vertex data as a dynamically growing Array of Structures (AOS). Furthermore, individual structures in this array can be freed and reclaimed through the *vertex queue*.

Depending on the graph type, vertices may require different parameters. As the memory management is not bound to a specific vertex size, each vertex can hold as many parameters as the application requires. Allocation of new vertices can be achieved in *O(1)*, the procedure first queries the *vertex queue*, thereby reusing freed indices from previous deallocations. If the queue does not hold any available data, we simply increase the dynamic array using an atomic fetch-and-add (ADD) on the vertex array size. Deleting a vertex includes deleting all edges referencing this vertex and returning its index to the *vertex queue* for later reuse. Keeping all vertex data next to another in memory has the advantage that simple indices can be used to reference vertices. The vertex' identifier used on the CPU is not bound to the memory location the vertex is stored at, we report a mapping between the host identifier and the device identifier back to the CPU. Additionally, when storing vertices sequentially, algorithms iterating over vertices show an efficient memory access pattern and better caching behaviour.

2) Edge Data: Allocating individual vertices is reasonable as there is usually no direct commonality between different vertices and memory requirements can be kept as low as possible. This strategy makes less sense for edges as there are usually many edges originating from the same vertex, which will often be iterated sequentially. Thus, edge data is placed on *pages* of a fixed size and multiple pages form a linked list of edges for every vertex, to dynamically adjust the size of the adjacency. This approach can be seen as a combination of a linked list and an adjacency array, yielding memory locality for edges within a page. At the same time, this strategy avoids reallocation of the whole adjacency if augmentation is required, by simply adding/removing a page to/from the linked list.

Pages can also be deallocated by returning a free page index to the *page queue* for later reuse. The page size forms a trade-off between overallocation and efficiency. A smaller page allows for a tighter bound closer to the actual number of adjacencies of a vertex, while a larger page size allows for more efficient traversal of the edges. At the same time, a too small page size also increases the number of pointers to the next page (we simply use the last four bytes on each page). Thus, the most suitable page size is application dependent and can be chosen to fit different scenarios. For all our experiments, we chose a page size of 64 *Bytes* (which coincides with the memory alignment of *cuSTINGER* and provides a good balance between performance and overallocation per adjacency for simple graphs). For the adjacency data itself, we support two memory layouts, of which either may achieve better performance depending on the traversal characteristics of the graph algorithms. If multiple properties per edge are required, the AOS approach provides better memory access characteristics. On the other hand, if just a single property is queried, using SOA is advantageous. Such properties include at least the destination vertex (simple graphs), weights (weighted graphs) and a timestamp (semantic graphs). For simple graphs, AOS and SOA are identical.

D. Vertex Updates

It is typical for graph structures to refer to vertices by their indices in memory, which alleviates look up procedures to locate vertices. This increases the cost of updates as a mapping procedure is required that maps an arbitrary vertex identifier on the CPU to an index on the GPU. Moreover, deleting vertices also has to be reflected in the adjacency data by removing all entries referencing said vertices. While edges are organized as a linked list of pages which support locking, all vertices are organized in the same pool of memory and need to be updated in parallel to achieve high performance.

1) Vertex Insertion: Vertex insertion is based on a four step approach to achieve parallel insertion, starting by sorting the update data batch, as can be seen in Algorithm 1. The next two steps are concerned with duplicate checking while the fourth performs the insertion itself. Duplicates can occur within a batch of to-be-inserted vertices and with vertices already present in the graph. Duplicates with the graph are non-trivial to find due to the mapping between CPU and GPU vertex identifiers. It would be very inefficient to search the entire GPU vertex structure for each to-be-inserted vertex. Thus, we propose a *reversed* duplicate check with the graph vertices. Given that the batch of to-be-inserted vertices is already sorted, searching in the batch is rather efficient. Thus, we start one thread for each graph vertex, which looks up its mapping from GPU to CPU identifier and performs a binary search on the sorted to-be-inserted vertices. If a duplicate is found, it is simply marked in a helper data structure to not hinder the subsequent checking step. Next, duplicate checking within the batch is performed and one thread for each entry is started. Each thread checks its batch successor and if a duplicate is found, simply marks it as a duplicate directly in the batch and continues as long as it finds duplicates in successive order. As the batch is sorted, this leaves only the first element of multiple duplicates remaining. After both steps, the helper data structure is synchronized with the edge update batch, removing duplicates with the graph from the batch as well. The actual vertex insertion process is straightforward: The framework starts by acquiring a new device index and a new page index for each valid vertex update.

Algorithm 1: Vertex Insertion

```
   Data: Vertex Update Batch
   Result: Vertices inserted into graph
1  Copy Vertex updates onto stack;
2  if sorting_enabled then
3      thrust::sort(vertex_updates);
4  d_duplicateCheckingInBatch (vertex_updates);
5  d_reverseDuplicateCheckingInGraph (vertex_updates, graph);
6  d_vertexInsertion (vertex_updates, graph);
7  Copy mapping back to host;
8  if sorting_enabled then
9      Copy vertex updates back to host;
```

Both first contact the respective queues for previously deleted indices. If a queue is empty, the memory manager supplies fresh indices. Then, the vertex is set up using the update data and the adjacency page is inserted. Finally, the new mapping from host identifier to device identifier, *i.e.*, each vertex' position in the vertex array, is reported back to the host.

2) Vertex Deletion: As each deletion procedure performs a CAS on the host identifier, only one thread will retrieve a valid identifier and continue the procedure, alleviating the need for duplicate checking. Contrary to vertex insertion, deleting a vertex not only alters vertex management data, but also has implications on adjacencies. This results from the fact that other vertices can reference said vertex by possessing an edge to it. These references have to be deleted from the graph structure as well and the holes left by these deletions have to be compacted in a separate procedure, as can be seen in Algorithm 2.

In case of an *undirected* graph, these references can be deleted directly in the deletion procedure, as each adjacency element has a dual that can directly be found by simply swapping source and destination of an edge. The procedure iterates over the adjacency of the to-be-deleted vertex and for each edge it removes the dual in the corresponding adjacencies. As no duplicates are present in the adjacencies, this deletion can even be performed without locking. If, on the other hand,

the graph is *directed*, the deletion procedure is not as straight forward and we use a multi step approach. For a directed graph arbitrary vertices may reference a to-be-deleted vertex. Thus, in a first step, we only return the pages allocated for the vertex to the page queue. The update data batch is once again sorted to speed up the following step. Once again, we propose a *reversed* deletion process similar to the *reversed* duplicate check, starting a worker per adjacency and searching each edge in the sorted update batch, which is once again rather efficient.

Algorithm 2: Vertex Deletion

```
Data: Vertex Update Batch
Result: Vertices deleted from graph
Copy Vertex updates onto stack;
if sorting_enabled then
    thrust::sort(vertex_updates);
d_vertexDeletion (vertex_updates, graph);
if graph_is_directed then
    d_reverseDeleteVertexMentions (vertex_updates, graph);
d_compaction (graph);
Copy mapping back to host;
```

After the actual deletion, the framework still has to perform compaction on the adjacencies. To avoid unnecessary locking during this step, the actual clean up is performed in a separate kernel by iteratively moving edges from the back to empty positions in the adjacency (or moving edges to the front consecutively to respect sort order). Again, using more than a single thread for this operation can increase performance. Finally, the now free vertex index is returned to the *vertex queue* and the mapping change is reported back to the host.

E. Edge Updates

Edge updates are considered a common operation for dynamic graphs. In *faimGraph* update information is considered to be made available to the framework by either the CPU-side and successively copied to the GPU or directly in a GPU buffer. The update procedure runs independently on the GPU in either case. A 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. Similarly to *aimGraph*, vertex structures hold a lock and update threads can lock each adjacency list before altering it to gain exclusive access. However, *faimGraph* adds support for multiple coordinated threads to alter adjacencies, which is preferable when scanning larger adjacency size. As coordinated threads need to communicate, we either use cooperative thread blocks or warps (groups of threads executing on the same SIMD unit). We call this kind of strategy an *update-centric* approach, as each update is mapped to an individual *worker* (thread/warp/block).

Locking strategies work well if the update pressure is not particularly high (updates are distributed over the graph well), the average size of the adjacencies is rather small (less than ≈ 25 according to our experiments), and the graph is well balanced. If updates in a batch favour a smaller set of vertices, the overhead introduced by locking as well as

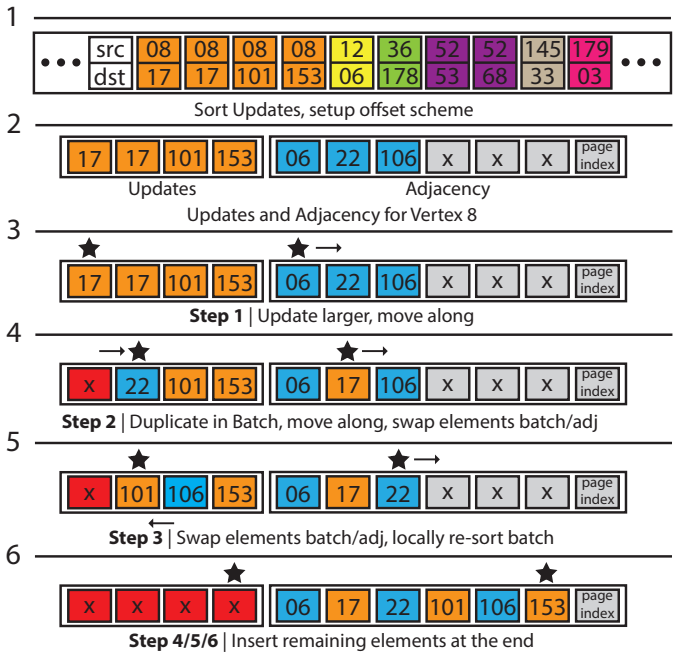

Fig. 2: Example for sorted insertion with duplicates in batch.

multiple adjacency traversals becomes a serious bottleneck. Thus, we propose a new update strategy that avoids locking by coordinating the update efforts beforehand: the *vertex-centric* approach. It devises an offset scheme to start a worker per vertex which is affected by updates. In this way, exclusive access to each adjacency is guaranteed and waiting on locks is avoided. Different update implementations can also be mixed for consecutive update calls to deal with changing requirements.

1) Edge Insertion: Our *vertex-centric* edge insertion splits the insertion process into three steps. First, an offset scheme is constructed: We sort the insertion requests according to the source vertex in-place. A following prefix sum determines the offset of each specific source vertex in the sorted array and the number of updates that will be performed for a specific adjacency. Second, duplicate checking is performed, which—if activated—makes sure that edges are only added to the graph once. To this end, the insertion requests are compared to the already existing graph and to the other requests in the sorted array. Third, one worker per affected vertex is started, which adds the edges to the end of the adjacency lists. If there is still sufficient space on the last page of the adjacency list, the edges are added to this page, otherwise additional pages are queried first from the *page queue*, if empty a new page index is supplied by the memory manager. This significantly reduces the update time and inserting millions of edges can be performed in a matter of milliseconds.

Furthermore, building on this approach it is also possible to respect sort order when inserting new vertices. During insertion, as can be seen in Figure 2, a combined sweep over the to-be-inserted and already present edges is then sufficient to insert the data: If an edge must be inserted before the end of the adjacency list, we simply swap the edge currently in this slot

with the update edge and merge the replaced edge into the insertion requests, hence the sorting effort is constrained to the update data targeting this specific vertex. Thus, when the end of the adjacency is reached, the remaining to-be-inserted edges can be placed at the back. Note that a sorted adjacency does not require a separate duplicate checking, as the entire existing adjacency is scanned during insertion anyway. Nevertheless, the determining factor for performance remains update pressure and adjacency traversal. High update pressure and longer traversal lend themselves to the *vertex-centric* approach, while, otherwise, *update-centric* provides better performance.

2) Edge Deletion: Vertex-centric edge deletion starts with the same sorting and prefix sum steps as *vertex-centric* edge insertion. Duplicate removal is not necessary for edge deletion, as edges can only be removed once. When a to-be-removed edge is found, we simply copy the last edge from the adjacency list over the edge to avoid holes in the list. To respect sort order we iteratively shuffle all remaining edges to the front, overwriting the to-be-deleted elements in the process. Again, the entire operation is performed in a single sweep over the edge data. If pages are left empty after the compaction step, they are returned to the *page queue*.

IV. Evaluation

In this section, we evaluate the performance of *faimGraph* and compare it to *aimGraph* and the publicly available *cuSTINGER*, as well as to *Hornet* and *GPMA* wherever possible. The performance measurements were conducted on a NVIDIA Geforce GTX Titan Xp (12 GB V-RAM), and an Intel Core i7-7770. The graphs used are listed in Table I. They represent a cross section of different problem domains and were taken from the *10th DIMACS Graph Implementation Challenge* [22].

A. Memory footprint

One of the biggest differences between *faimGraph* and previous approaches is memory consumption and memory footprint over time. Although *faimGraph* starts with a larger allocation as it manages memory directly on the device, the actual memory footprint within the framework (especially over time) is lower compared to previous approaches and all memory allocations are facilitated directly through the framework without host intervention. Since the cost of reinitialization is negligible in most cases (due to the relative addressing within the pool which allows the usage of just two `memcpy`'s to reinitialize), even the initial allocation can be chosen conservatively. *cuSTINGER* performs sequential allocation calls from the CPU to allocate the management data and all individual edge blocks in the initialization procedure. Especially for graphs with millions of vertices, this is a significant overhead, compared to the single allocation in *faimGraph*.

Furthermore, due to the overhead associated with reallocation, *cuSTINGER* uses over-allocation to reduce the runtime cost for edge updates. *faimGraph* locates all its data by combining an efficient indexing scheme and reinterpreting memory on the fly. This way, the same functionality can be achieved with significantly less memory. Table I notes the respective memory footprints within the framework for *faimGraph*, *cuSTINGER* as well as *GPMA*. The difference is most significant for high numbers of vertices (*e.g.*, `europe` (14) with 4GB vs 7GB) for *cuSTINGER*, but also for large adjacencies due to overallocation (*e.g.*, `nlpkkt200` (12) with 2GB vs 4GB for *cuSTINGER* or `audikw1` (4) with 250MB vs 420MB for *GPMA*) for both *cuSTINGER* and *GPMA*. *GPMA* performs well for very sparse graphs as it stores no additional vertex properties (*e.g.* number of edges per adjacency), but experiences significant overhead for denser graphs as each edge has to store both source and destination as part of the *PMA*.

B. Memory usage evaluation

faimGraph's memory management scheme allows for reuse of memory over time. This is especially crucial for long-term use cases, where certain areas of the graph grow and shrink significantly. Both *aimGraph* and *cuSTINGER* hold the maximal allocation state in memory, meaning that once allocated, memory stays with its vertex. Hence, after prolonged usage the allocated memory resources do not reflect the actual memory requirements and may even lead to system failure over time. To test long term use, we use three different test cases: The *Uniform* test case performs successive edge insertions and deletions derived from a uniform distribution. *Random* performs the same operations, whereas each round is randomly chosen to be either insertion or deletion. The memory footprint for these tests is shown in Table I. *aimGraph* is more efficient in all cases compared to *cuSTINGER*, requiring between 12% to 45% less memory. *faimGraph* reduces the memory consumption further to 27% to 52% less memory compared to *cuSTINGER*. The *Sweep* testcase highlights the behaviour for strongly volatile graphs. Each update round targets a set of 100 vertices with a batchsize of 1.000.000, where a set of edges is first inserted and then deleted again. Each update targets a successive set of source vertices. Performance is measured in rounds (how long can this procedure be repeated before the system goes out of memory). As shown in Table I, *faimGraph* can run to completion for all graphs as the memory footprint after each round is mostly equal to the initial state. *aimGraph* and *cuSTINGER* fall significantly behind and only manage to complete all rounds within the 12 GB of memory for graphs that only require less than 100 MB by itself—the memory that *faimGraph* returns to after every deletion step during the sweep test. For these small graphs, our tests thus revealed a memory increase of more than two orders of magnitude above the necessary. This clearly underlines that fully dynamic memory management is essential in highly volatile problem domains. Using a large semi-continuous array, *GPMA* should able to reuse freed memory in the *Sweep* testcase as well. Unfortunately, since memory is localized, significant rebalancing might be required and subsequently, performance would be penalized. The current implementation does not handle duplicates within the update batch, but even correcting for that unfortunately still accumulates memory and we were not able to trace the source of this issue. As there is only limited information available about *Hornet* at this point, we are unsure of its

internal behaviour. In any case, it seems to be an improvement over its predecessor *cuSTINGER*.

C. Initialization

The autonomous approach to memory management on the GPU pays off during initialization. *faimGraph* distributes memory to individual vertices fully in parallel and the single GPU memory allocation drastically reduces allocation overhead. *cuSTINGER* performs sequential iterations over all vertices to allocate its adjacencies. In all tested scenarios (*cf.* Table I), *faimGraph* is able to outperform all other approaches by a significant margin. The same is true for reinitialization with increased size, which is even faster than pure initialization due to our favorable relative indexing setup. The discrepancy in performance (up to two orders of magnitude) is greatest for graphs with a large number of vertices, like `germany` (10) and `europe` (14). But still in small dense graphs, *e.g.*, `ldoor` (4), the speed-up is one order of magnitude. Performance overhead for reinitialization with 105% of the conservative allocation size is displayed in Table I as well.

D. Edge Updates

Figure 3 and 4 show the insertion and deletion performance for *faimGraph* as well as *cuSTINGER*, *GPMA* and *Hornet* for uniform and focused update distributions. *faimGraph* utilizes the conservative memory allocation with 50% additional pages in these tests, due to efficient memory re-use none of the cases experienced reinitialization.

1) Edge Insertion: The clearest performance difference for edge insertion can be observed for graphs with small to medium sized adjacencies, which can be attributed to two factors: *cuSTINGER* performs an additional indirection step to follow the data structure and employs overallocation to reduce the need for reallocation. For larger graphs, the probability for insertions to hit the same vertex is smaller and thus, reallocation does not happen at all for *cuSTINGER*. For smaller graphs, on the other hand, the performance numbers quite clearly reflect the overhead introduced by reallocation procedures. *faimGraph* on the other hand directly interprets memory as required and does *not* employ overallocation. The *vertex-centric* approach is not particularly well suited to uniform updates with low update

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
Type	Road	Citation	Citation	Matrix	Matrix	Triang.	Geom.	Simu.	Triang.	Road	Matrix	Matrix	Matrix	Road
—V—	115k	227k	299k	952k	943k	1.04M	1.04M	5.82M	8.38M	12M	3.5M	16.24M	27.99M	50.91M
—E—	239k	815k	1.95M	45.57M	76.71M	3.14M	6.89M	8.73M	25.16M	24.74M	93.3M	431.9M	746.4M	108.1M
faimGraph														
Initialization (ms)	1.16	2.47	2.61	58.11	102.92	6.38	11.60	20.21	45.88	32.78	100.03	459.36	792.88	145.16
Initialization (MB)	9.18	20.41	26.44	244.66	355.53	84.0	99.98	466.59	672.01	925.16	494.62	2277.3	3929.8	4078.4
Reinit. 105% (ms)	0.36	0.58	0.613	6.10	8.90	1.41	2.38	5.18	9.21	9.42	10.49	27.23	36.04	26.75
Uniform (MB)	41.97	49.97	54.44	235.30	359.99	103.45	121.97	467.16	675.01	925.17	544.83	2349.1	3980.7	4078.4
Random (MB)	50.73	60.06	65.03	257.22	382.80	114.98	138.13	467.74	676.98	925.18	569.59	2372.6	3996.3	4078.4
Sweep (Rounds)	all	all	all	all	all	all	all	all	all	all	all	all	all	all
Queries (ms)	2.61	3.26	3.12	4.58	5.73	3.19	3.21	3.28	3.23	3.23	3.28	3.23	3.24	3.21
V. Insertion (ms)	1.99	2.09	2.15	2.82	2.80	2.85	2.87	6.75	9.42	11.33	4.87	15.19	25.63	45.68
V. Del. (UD) (ms)	1.49	2.29	2.82	12.31	29.63	2.90	4.01	7.07	10.42	11.54	22.31	52.98	74.27	43.94
V. Del. (D) (ms)	1.26	2.36	2.02	11.87	24.54	2.89	4.03	8.67	18.95	15.75	20.84	83.15	144.15	69.17
aimGraph														
Uniform (MB)	49.13	59.82	65.68	267.66	392.69	122.17	149.99	467.83	678.40	925.18	608.64	2430.5	4038.4	4078.4
Random (MB)	61.14	72.35	78.11	284.52	411.06	136.88	163.31	468.22	678.88	925.19	625.33	2410.1	4014.8	4078.4
Sweep (Rounds)	all	all	all	2547	2518	2584	2580	2481	2429	2367	2473	2033	1589	1523
cuSTINGER														
Initialization (ms)	94.48	191.32	254.37	966.58	1013.4	982.69	944.55	6177.8	9286.9	12752	3848.9	20907	-	64124
Initialization (MB)	16.61	36.81	47.71	369.99	548.99	152.03	195.34	844.32	1216.1	1674.1	930.19	4292.8	-	7380.1
Uniform (MB)	66.86	87.97	100.02	381.70	553.93	216.32	247.44	855.76	1264.2	1675.6	948.15	4295.2	-	7380.1
Random (MB)	69.52	89.11	99.78	378.38	551.27	208.44	245.41	847.19	1236.7	1674.2	936.91	4293.6	-	7380.1
Sweep (Rounds)	all	2060	2057	1999	1951	2049	2046	1898	1824	1732	1892	1262	-	593
GPMA														
Initialization (ms)	34.27	55.24	52.92	224.81	387.50	50.61	86.08	121.14	308.22	181.23	467.94	-	-	782.85
Initialization (MB)	4.68	26.92	30.87	422.51	702.64	70.25	137.68	232.92	562.04	372.78	885.78	-	-	1634.84
Uniform (MB)	209.79	106.16	105.84	-	731.64	90.38	157.33	-	587.97	400.94	996.23	-	-	1650.84
Random (MB)	282.73	299.92	257.89	-	1110.37	294.00	286.75	-	843.84	621.70	1133.00	-	-	1915.54
Sweep (Rounds)	176	73	68	11	19	264	29	28	136	49	60	-	-	38
Hornet														
Initialization (ms)	4.20	10.73	13.18	168.59	207.06	45.99	61.72	300.87	468.20	434.94	755.97	53k	230k	4410.6
Sweep (Rounds)	all	all	all	6504	3140	3998	2899	5862	3314	1641	5701	4715	3955	1447

TABLE I: Performance measurements for *cuSTINGER*, *Hornet*, *GPMA*, *aimGraph* and *faimGraph*, including initialization time and overall timings for a complete test set as well as memory evaluation on three test cases on the graphs *luxembourg* (1), *coAuthorsCiteseer* (2), *coAuthorsDBLP* (3), *ldoor* (4), *audikw1* (5), *delaunay_20* (6), *rgg_n_2_20_s0* (7), *hugetric-00000* (8), *delaunay_n23* (9), *germany* (10), *nlpkkt120* (11), *nlpkkt200* (12), *nlpkkt240* (13) and *europe* (14)

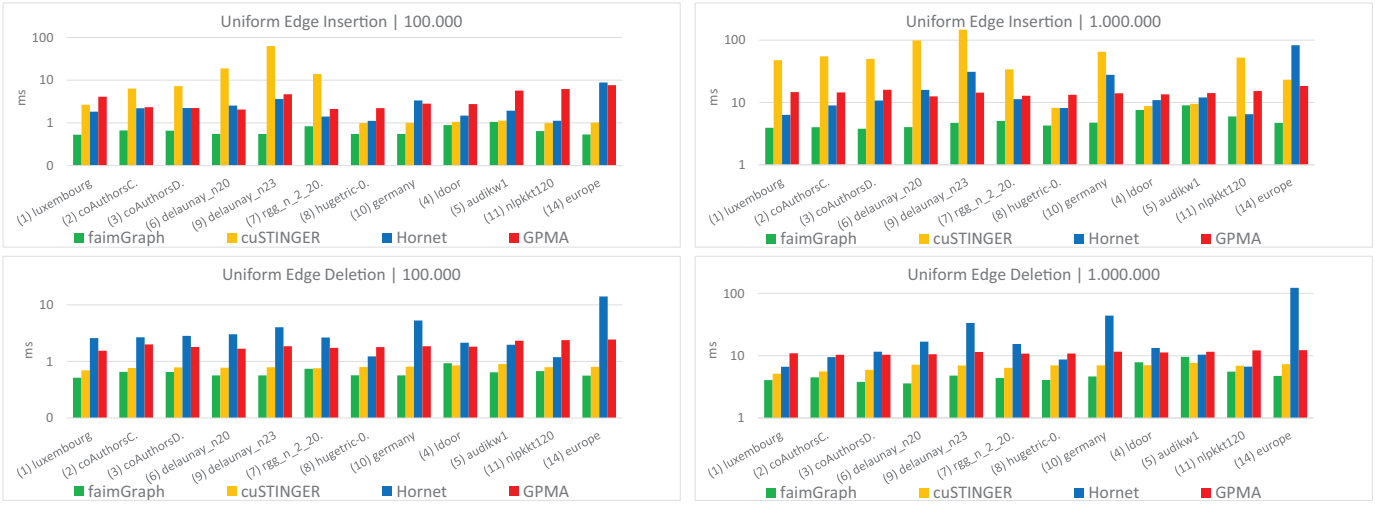

Fig. 3: Update timings in ms on a log-scale for uniform edge updates, using batch sizes of 100.000 and 1.000.000, showing the *faimGraph* implementation as well as *cuSTINGER*, *Hornet* and *GPMA*.

pressure. This results from the excessive duplicate checking needed in this case. The *sorted* approach performs exceptionally well, especially for small to medium sized adjacencies, as the benefits derived from sorted adjacencies and updates during the procedure outweigh the re-sorting effort. Thus, keeping sorted adjacencies actually introduces no to hardly any overhead. Only for large adjacencies the performance scales negatively with the increased memory access needed by the sorting procedure. Increasing the update pressure (focusing the updates on a range of 1000 vertices, which sweeps over the graph throughout the test), as shown in Figure 4, yields consistently good results for *faimGraph*, even with the *update-centric* approach. Although the locking overhead is clearly visible in all cases, locking still outperforms *cuSTINGER* in all cases as the reallocation procedures are handled more efficiently. Both *vertex-centric* approaches outperform the *update-centric* approach due to reduced overhead while traversing the adjacency and the removal of locking. *faimGraph* outperforms *cuSTINGER* by a factor of $1.1\times$ - $114\times$ / $1.1\times$ - $31\times$ for both batch sizes with uniform edge insertion; focussing updates on a smaller range of vertices yields a speed-up between $25\times$ - $185\times$. Similarly, the speed-up of uniform updates compared to *GPMA* is $2.5\times$ - $14\times$ / $1.6\times$ - $4.2\times$; with higher update pressure the difference is $2.1\times$ - $5.4\times$. Compared to *Hornet*, the speed-up achieved for uniform updates is $1.6\times$ - $16.5\times$ / $1.1\times$ - $17.6\times$, using higher update pressure updates *Hornet* performs slightly better in one case, the overall speed-up falls between $0.93\times$ - $6.48\times$.

In summary, it can be noted that *cuSTINGER* performs well when it works within its overallocation boundaries, but otherwise drops significantly. *GPMA* performance is very uniform independent of the sparsity of the graph, but is slower overall. *Hornet* performs well if the source vertex range modified is small, but falls behind significantly for large source vertex ranges in sparse graphs. The best *faimGraph* strategy is always faster than *cuSTINGER*, which can easily be selected based on the update pressure. Higher update pressure clearly favours our new *vertex-centric* approaches.

2) Edge Deletion: In case of deletions, the performance difference is slightly less pronounced and much more consistent compared to the insertion process. This is not surprising as edge deletion is very straight forward in *cuSTINGER*, as memory is not freed and duplicate checking is not necessary. *faimGraph* on the other hand additionally performs compaction and frees non-needed pages, which introduces additional overhead. However, due to the smaller memory footprint and more efficient implementation, *faimGraph* still outperforms *cuSTINGER* in 10 of the 12 test cases for uniform deletion (Figure 3) with a performance difference between $0.92\times$ - $1.4\times$ / $0.8\times$ - $1.9\times$. Compared to *GPMA*, *faimGraph* is always faster with a speed-up of $1.9\times$ - $3.6\times$ / $1.4\times$ - $2.9\times$. The same is true for *Hornet*, the difference here is $1.7\times$ - $25\times$ / $1.1\times$ - $25.9\times$. Sorting again hardly reduces performance compared to the *vertex-centric* approach unless large adjacencies need to be moved as in `ldoor` (4) and `audikw1` (5). For larger graphs,

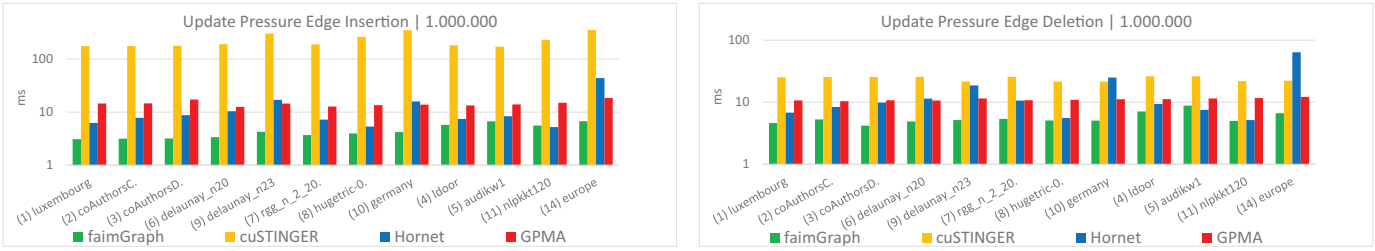

Fig. 4: Update timings in ms on a log-scale for pressure edge updates, using a batch size of 1.000.000, half targeting a range of 1000 vertices, sweeping over the graph, the other half uniformly distributed over the graph.

with few updates per vertex, the locking strategy performs best. For high update pressure (Figure 4), both *vertex-centric* approaches perform best in all test cases, clearly outperforming both the *update-centric* approach as well as *cuSTINGER*, with a performance difference between $3\times$ - $6\times$, this difference is $1.3\times$ - $2.6\times$ compared to *GPMA* and $0.85\times$ - $9.7\times$ to *Hornet*.

E. Vertex Updates

Previous dynamic graph frameworks, such as *cuSTINGER* and *aimGraph*, are only *partially-dynamic*. Their SOA approach for vertices makes it difficult to efficiently update vertices. Contrary, *faimGraph*'s AOS approach and index queues allow for fully dynamic vertex insertion as well as deletion.

1) Vertex Insertion: Table I shows the timings for vertex insertion with a batch size of 100.000. The actual insertion process for all tested graphs stays below $1ms$, the overall timing with duplicate checking stays below $50ms$ for all tested graphs. Although our *reverse* duplicate checking increases performance significantly, duplicate checking still forms the bottleneck for larger graphs. As the duplicate checking involves all graph vertices, the execution time is proportional to the number of graph vertices. For the tested graphs, *faimGraph* can facilitate 2 - 50 million vertex insertions per second.

2) Vertex Deletion: Vertex deletion is more complicated than vertex insertion, as all references to the vertex in the graph must also be deleted. Table I shows the performance numbers for vertex deletion in case of *undirected graphs* (UD) as well as *directed graphs* (D). In both cases the same graphs are used, *i.e.*, undirected graphs can be treated as directed graphs. The performance difference is only due to the different deletion strategies that can be employed for the two cases. For undirected graphs, the vertex mentions are deleted directly in the deletion kernel, which prolongs the vertex deletion stage but does not require an additional stage afterwards. For *directed* graphs there is an extra step involved, as it is not directly obvious where the directed edges might reside in memory. This additional kernel once again profits from sorting the updates to utilize the reverse search pattern used to detect now invalid edges. The main difference can be observed for larger and denser graphs, as the search for references to deleted vertices is more costly as all vertices and their adjacencies have to be checked. Overall, the framework is able to handle between $1-50$ million vertex deletions per second, performance scales with both the number of vertices and edges present.

V. Algorithms

To evaluate the impact of our memory management data structure on algorithmic performance, we implemented triangle counting and PageRank [15] as two challenging algorithms on top of *faimGraph*. We compare our implementation to *cuSTINGER*, which includes the fast triangle counting by Green et al. [13] and a custom PageRank implementation as well as to *Hornet*. Unfortunately, *cuSTINGER*'s implementations did not run on recent hardware, thus we additionally include performance measurements for an NVIDIA GTX 780.

A. Work-balancing

One of the issues for graph frameworks is varying sparsity over the whole graph. Algorithms traversing these adjacencies may show significant imbalances. Thus, in addition to naïve implementations of the two algorithms, we introduce a work balancing scheme that, instead of launching a worker per vertex, calculates an offset scheme to locate individual pages in memory. This information is used to start one worker (thread/warp/block) per page per vertex. There is a clear correlation between the overhead introduced and the pages in memory. However, according to our experiments, the overhead stays small (between 0.3 - $1.5ms$ in our tests) and the benefits drastically increase with more pages in memory.

B. Static Triangle Counting - STC

The fast triangle counting algorithm [13] employed by *cuSTINGER* is based on a list intersection algorithm called **Intersect Path**. The algorithm operates on two stages of

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
Type	Road	Citation	Citation	Matrix	Matrix	Triang.	Geom.	Simu.	Triang.	Road	Matrix	Matrix	Matrix	Road
—V—	115k	227k	299k	952k	943k	1.04M	1.04M	5.82M	8.38M	12M	3.5M	16.24M	27.99M	50.91M
—E—	239k	815k	1.95M	45.57M	76.71M	3.14M	6.89M	8.73M	25.16M	24.74M	93.3M	431.9M	746.4M	108.1M
STC														
faimG. naïve (780)	0.077	21.36	35.56	258.4	1055.93	5.55	15.67	5.21	35.56	5.16	942.35	-	-	-
faimG. bal. (780)	0.70	9.08	11.79	280.02	1001.6	6.14	8.18	7.24	44.85	8.37	881.48	-	-	-
cuSTINGER (780)	9.71	35.83	45.72	379	1277.1	124.41	166.21	535.27	983.14	947.45	649.07	-	-	-
CSR (780)	7.42	20.94	26.86	260.03	635.8	98.59	140.75	390.43	771.06	698.38	327.43	-	-	-
faimG. naïve (Xp)	0.03	9.19	14.56	152.41	742.23	1.37	6.92	1.53	10.18	1.74	307.89	1413.9	2490.4	7.60
faimG. bal. (Xp)	0.64	3.57	4.90	87.26	402.56	1.82	7.34	2.34	11.34	3.26	197.80	923.98	1484.71	14.04
Hornet (Xp)	1.63	15.61	13.36	235.33	479.63	24.78	44.26	65.26	170.66	99.52	308.63	1111.5	1812.3	344.30
PageRank														
faimG. naïve (780)	0.54	1.59	1.01	16.18	31.4	1.97	4.04	8.39	14.28	8.31	26.51	-	-	-
faimG. bal. (780)	1.09	1.22	1.36	11.98	18.90	2.38	4.03	7.99	16.42	16.16	16.02	-	-	-
cuSTINGER (780)	0.88	1.94	2.62	20.25	30.18	6.06	10.31	26.21	42.43	47.96	57.53	-	-	-
faimG. naïve (Xp)	0.28	1.09	0.59	12.87	31.00	0.99	2.19	3.95	7.07	5.31	15.20	69.32	121.91	18.49
faimG. bal. (Xp)	0.30	0.32	0.31	2.67	5.07	0.65	0.84	3.10	4.79	4.74	4.21	18.94	32.67	17.63
Hornet (Xp)	0.48	0.28	0.32	3.58	5.83	0.80	5.26	3.76	5.63	5.86	5.54	25.08	43.29	23.60

TABLE II: Algorithmic performance measurements (average for one computation) in ms for *cuSTINGER*, *Hornet* and *faimGraph* (graph numbering identical to Table I) with a NVIDIA GTX 780 and a NVIDIA GTX TITAN X(p).

parallelism. The first stage balances the vertices on the multiprocessors and the second stage balances the adjacency access using different block sizes. The key search strategy of the algorithm is that a sorted adjacency allows for efficient binary search to identify triangles. *cuSTINGER* includes this implementation for its own data structure and for CSR.

Our naïve *faimGraph* implementation starts one worker for each vertex and iterates over the respective adjacency. It then checks for each pair of vertices in the adjacency, if this pair is connected. This checking stage is only performed, if the vertex, whose adjacency is examined, has the largest index in the triple under investigation. If a triangle is found this way, the triangle count is increased for all three vertices. By assuming a sorted (in ascending order) adjacency, this approach can even halt the procedure earlier, as soon as both vertices in the vertex pair under investigation are larger than the source vertex (in these cases a possible triangle will be entered by one of the other vertices). The balanced *faimGraph* implementation works similarly, but starts on worker per page per vertex, reducing the workload per worker. Performance numbers are recorded in Table II. Both *cuSTINGER* and *faimGraph* utilize the property that the adjacency is sorted to make comparison possible. *faimGraph* is able to significantly outperform *cuSTINGER* in all but *nlpkkt120* (11), which shows very long adjacencies. *faimGraph* can only partially derive an advantage from a sorted adjacency as an efficient search within a sorted array is only possible within page boundaries. Interestingly, our work balancing also outperforms the highly compact CSR format (using the fast triangle counting algorithm) in all but three cases. *faimGraph* is not well suited for naïve random adjacency access and thus triangle counting is one of the most challenging use cases for our data structure. Hence, a straight forward translation of the *Intersect Path* algorithm to *faimGraph* would also not show the same performance results as on a simple array structure. *faimGraph* is well suited even for memory intensive algorithms, if the average adjacency size does not grow incessantly, using *work balancing* even unbalanced graphs can be handled well. Overall, in 10 out of 11 cases, *faimGraph* has a performance lead between $1.25\times$ - $100\times$ over *cuSTINGER*, only falling behind in one test case. Compared to *Hornet*, *faimGraph* has a performance lead between $1.19\times$ - $57\times$ in all cases.

C. PageRank

PageRank [15] is a fairly straightforward algorithm. The algorithm has to traverse the adjacencies of all vertices and compute the contributions of all relationships for each vertex, similar to an *SpMV*. This means that every edge is touched exactly once, the same is true for every vertex. The only point of contention remains the *PageRank vector* itself. Table II shows the direct comparison between *cuSTINGER*, *Hornet* and the two (standard and balanced) *faimGraph* implementations. As PageRank has moderate memory access requirements and does not benefit from sorting, *faimGraph* is able to outperform *cuSTINGER* in all cases due to the more efficient memory access and footprint characteristics. Unbalanced and larger graphs once again profit from *work balancing*. Overall, *faimGraph* is able to outperform *cuSTINGER* by a factor of $1.5\times$ - $5.5\times$. The same is true compared to *Hornet*, but the performance difference is smaller overall in a range between $0.88\times$ - $6.2\times$.

VI. Conclusion & Future Work

faimGraph is a memory-efficient, fully dynamic graph solution with autonomous memory management directly on the GPU. Based on a queuing scheme, memory is fully reused within the system, reducing memory requirements by multiple orders of magnitude in the long run as well as memory fragmentation, permitting edge insertions and deletion according to arbitrary patterns. Thus, *faimGraph* can be safely used in real-world scenarios without threatening system failures due to out of memory. Furthermore, *faimGraph* is fully-dynamic, allowing for efficient vertex insertion and deletion at high rates, increasing access characteristics by efficiently reusing free vertex indices. Our *vertex-centric* update scheme allows lock-free edge updates, which increases performance by one order of magnitude under high update pressure. Edge updates can also respect sort order with little overhead.

faimGraph outperforms the previous state-of-the-art in all tested graphs in terms of edge update rate (up to $150\times$ higher update rate) as well as initialization time (up to $300\times$ faster). The framework can hold tens of millions of vertices and hundreds of millions of edges in memory. It is able to process up to 200 million edge updates and more than 300 million adjacency queries per second for the tested graphs. Vertex updates can also reach between $1 - 50$ million updates per second. To validate algorithmic performance of our data structure, we tested triangle counting and PageRank. Although *faimGraph* uses a more complicated data structure to allow for memory reclamation, it performs surprisingly well for the random access heavy triangle counting, outperforming *cuSTINGER* in all but one case. For PageRank, *faimGraph* showed the best performance in all cases.

In the future, we will expand *faimGraph* to multi-GPU systems, where challenging memory layout and balancing issues need solving. Furthermore, work distribution is required to go hand in hand with memory distribution, requiring foresight into algorithmic behaviour to achieve good performance. Another important topic is concurrent memory management and algorithm execution. For the first time, it is possible to perform both in parallel using autonomous memory management; we even support algorithms to directly manipulate the graph. However, both operations may still require communication and appropriate synchronization, which poses new scheduling challenges which we intend to solve in the future. Nevertheless, we believe that *faimGraph* is a first big step towards using GPUs for real-world, dynamic graph processing.

Acknowledgment

This research was supported by the German Research Foundation (DFG) grant STE 2565/1-1, and the Austrian Science Fund (FWF) grant I 3007. The GPU for this research was donated by NVIDIA Corporation.

REFERENCES

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, pp. 202–210, 2005.

[2] NVIDIA, "nvGraph," https://developer.nvidia.com/nvgraph, 2016, [Online; accessed 12-May-2017].

[3] L. SYSTAP, "BlazeGraph," https://www.blazegraph.com/, 2017, [Online; accessed 01-May-2017].

[4] 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.

[5] 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.

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

[7] M. Winter, R. Zayer, and M. Steinberger, "Autonomous, independent management of dynamic graphs on gpus"," in *2017 IEEE High Performance Extreme Computing Conference (HPEC '17)*. University of Technology, Graz, 2017.

[8] O. Green and D. Bader, "custinger: Supporting dynamic graph algorithms for gpus," in *2016 IEEE High Performance Extreme Computing Conference (HPEC '16)*. Georgia Institute of Technology, 2016.

[9] M. Sha, Y. Li, B. He, and K. Tan, "Accelerating dynamic graph analytics on gpus," in *International Conference on Very Large Data Bases 2018*. National University of Singapore, 2018.

[10] F. Busato, O. Green, N. Bombieri, and D. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," in *2018 IEEE High Performance Extreme Computing Conference (HPEC '18)*. Georgia Institute of Technology, 2018.

[11] 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.

[12] 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.

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

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

[15] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Computer Networks and ISDN Systems*. Stanford University, 1998.

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

[17] 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.

[18] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Catalyürek, "Betweenness centrality on gpus and heterogeneous architectures," in *6th Workshop on General Purpose Processor Using Graphis Processing Units*, 2013.

[19] 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.

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

[21] B. Kerbel, M. Kenzel, J. H. Mueller, D. Schmalstieg, and M. Steinberger, "The broker queue: A fast, linearizable fifo queue for fine-granular work distribution on the gpu," in *International Conference on Supercomputing 2018*. University of Technology, Graz, 2018.

[22] 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.

APPENDIX

A. Abstract

The following appendix describes the framework setup, how it can be used and how individual tests can be run.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Both STC and PageRank can be tested using the executables *STCaimGraph* as well as *PageRankaimGraph*
- **Compilation:** CMake 3.2 to setup, CUDA 9.1 and C++14 compliant compiler (tested with *gcc-6* and *MSVC 19.00.24225.1*)
- **Data set:** Use graphs from 10th DIMACS Implementation Challenge
- **Hardware:** Reasonably recent GPU hardware from NVIDIA (Kepler or onward, tested on an NVIDIA GTX 780 and an NVIDIA GTX TITAN Xp)
- **Execution:** Call corresponding executable for each testcase
- **Output:** Output is produced in the output stream
- **Experiment workflow:** Provide the corresponding XML configuration to the right executable
- **Experiment customization:** XML configuration files can be adapted for different batchsizes, different update strategies, different page sizes etc.
- **Publicly available?:** Available at BitBucket

2) How software can be obtained: The framework can be downloaded from BitBucket.

3) Hardware dependencies: Recent GPU hardware from NVIDIA (e.g. Kepler GPUs and onward), tested on an NVIDIA GTX 780 as well as a NVIDIA GTX TITAN Xp. Remaining system specifications should have a comparatively small impact on performance, performance was tested on an Intel Core i7-7770 paired with 16 GB RAM.

4) Software dependencies:

- CUDA 9.1
- C++14 compliant compiler, tested on:
  - gcc-6
  - MSVC (Visual Studio 2017)
- CMake 3.2 or higher

5) Datasets: The framework currently supports graphs in a CSR format as provided by the 10th DIMACS Implementation Challenge, all graphs used for performance evaluation where downloaded from this website.

C. Installation

Project setup uses CMake to configure the project itself, on Linux simply enter (the same can be accomplished on Windows using a graphical user interface):

```
mkdir build
cd build
cmake ..
make -j4
```

This should set up the framework itself and build the framework and create executables for all the different testcases, this includes

- *mainaimGraph*
  - Can be used to test edge update performance
- *reInitTC*
  - Can be used to test the reinitialization procedures of the framework
- *STCaimGraph*
  - Can be used to test triangle counting performance for faimGraph
- *PageRankaimGraph*
  - Can be used to test PageRank performance of faimGraph
- *continuousTCaimGraph*
  - Can be used to query memory information for long term use cases (includes uniform updates, random updates as well as sweep updates)
- *dynamicVerticesMain*
  - Can be used to test dynamic vertex updates (vertex insertion & deletion for undirected and directed graphs)
- *concurrentTCaimGraph*
  - Can be used to test concurrent edge updates (edge insertion/deletion either in the same kernel or in separate streams)
- *queryTCaimGraph*
  - Can be used to test the edge query ability of the framework

Each testcase can be configured by an *XML*-File that is passed to the executable, in this file the user can configure which graphs to test, which device to use, how much memory should be allocated as well as most parameters of *faimGraph*, including

- pagesize (how large are individual pages in Bytes)
- memorylayout (Memorylayout on pages *AOS* vs *SOA*)
- updatevariant (update centric vs. vertex centric vs. vertex centric sorted)
- rounds + updaterounds ($10 \cdot 10$ in the standard configuration, rounds includes initialization of the framework, updaterounds just one round of insertion/deletion)
- stack/queue sizes (in Bytes / in indices)

Different configuration files are already provided for the different testcases, furthermore the user can inspect the *Configurationparser* class for all customizable parameters. To start a testcase (e.g. to test edge update performance with the vertex centric approach using a small test set), simply update the paths to the graphs to test in the configuration XML file and call

```
./mainaimGraph vertexcentricsmalltest.xml
```

D. Experiment workflow

The following section describes how to perform the different experiments used to evaluate the performance of *faimGraph*.

1) Vertex Update Performance: Vertex update performance can be tested using the *dynamicVerticesMain* executable, performance influencing factors include the *pagesize*, *batchsize* as well as the *graph directionality*.

2) Edge Update Performance: Edge update performance can be tested using the *mainaimGraph* executable, it can be started in the following modes

- **Standard**: Updates using locking, can be specified to use Threads or Warps per update
- **VertexCentric**: Updates using preprocessing to start Threads per affected vertex
- **VertexCentricSorted**: Similar to vertex centric, but respects sort order in the adjacency when updating

Performance influencing factors include the *pagesize*, *batchsize* as well as the *update strategy*.

3) Edge Query Performance: Connection queries can be tested using the *queryTCaimGraph* executable, the only performance relevant factors are the *pagesize* as well as the *batchsize* for the queries.

4) Algorithmic Performance: Algorithmic performance can be tested using the *STCaimGraph* and *PageRankaimGraph* executables respectively, the performance relevant factors include the *pagesize* and if work balancing should be used (this is currently configured directly in the main file and requires a recompile upon change).

5) Memory Performance: Memory performance can be evaluated using the *continuousTCaimGraph* executable, the different TCs can be selected by a compile flag up top the main file, performance relevant factors once again include the *pagesize* as well as the *batchsize*.

E. Evaluation and expected result

The project can be build both on Linux and windows, the specific performance numbers were gather on Linux 16.04 *Xenial* LTS with CUDA 9.1 and gcc-6, using an NVIDIA GTX TITAN Xp (as well as an NVIDIA GTX 780 for parts of the algorithmic evaluation) and an Intel Core i7-7770 with 16GB RAM. The individual testcases and how to reproduce the results is already described in Section D.

F. Experiment customization

Each experiment can be configured by altering the corresponding *XML* file or the test setup, the individual main files to the corresponding executables should give guidance on how to individually construct different experiments.

G. Notes

As this is still a research project, the current state is still prone to *misconfiguration*, hence it is possible to provide the framework with invalid or ”bad” values without a warning(e.g. setting the *queue* size to 0, hence loosing access to all returned indices). The provided configuration files should provide a guideline on how to set up the project to perform as intended, if questions do arise, it would be highly appreciated to seek contact with the authors for clarification.