Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs

Federico Busato*, Oded Green†, Nicola Bombieri* and David A. Bader†
* Department of Computer Science, University of Verona - Italy
† Computational Science and Engineering, Georgia Institute of Technology- USA

***Abstract*—Sparse data computations are ubiquitous in science and engineering. Unlike their dense data counterparts, sparse data computations have less locality and more irregularity in their execution, making them significantly more challenging to parallelize and optimize. Many of the existing formats for sparse data representations on parallel architectures are restricted to static data problems, while those for dynamic data suffer from inefficiency both in terms of performance and memory footprint. This work presents *Hornet*, a novel data representation that targets dynamic data problems. *Hornet* is scalable with the input size, and does not require any data re-allocation or re-initialization during the data evolution. We show a *Hornet* implementation for GPU architectures and compare it to the most widely used static and dynamic data structures.**

***Index Terms*—Dynamic Graph Structures, GPU Computing, Graph Analytics**

I. Introduction

Dynamic sparse data applications are now ubiquitous and can be found in many domains. *Dynamic* refers to the fact that the data is changing at very high rates. The sparsity of the data has led to the development of several data representations, which are common for both problem formulations: Compressed Sparse Row *(CSR)*, Coordinate (*COO*), Compressed Sparse Column (*CSC*), and ELL (*Ellpack*). Unlike a dense adjacency matrix, which may be potentially filled with "0"-values, these formats avoid storing these trivial values. As such, these data-structures are cost-effective in terms of memory yet lack flexibility to support growth.

In this context, even though some attempts have been recently made to design a data structure that is scalable, high-performing, and flexible enough to support rapid updates [1]–[4], these are unable to meet all criteria.

This paper presents *Hornet*, a data structure for efficient computation on *dynamic sparse graph and matrices*. *Hornet* can grow to very large sizes without requiring any data re-allocation or re-initialization during the whole dynamic evolution of data.

Hornet outperforms dynamic graph data structures at the state of the art on several fronts: *Hornet* provides better memory utilization than AIM [4] and cuSTINGER [2], faster initialization (from $3.5x$ to $26x$ than cuSTINGER), and faster update rates (over 200 million updates per second); *Hornet* uses a small fraction of the memory that AIM requires and about $5x$-$10x$ less memory than cuSTINGER. Compared to the static data structures, *Hornet* requires only 5% to 35% additional memory in contrast to CSR and, in average, 30% *less* memory than COO.

This paper presents a *Hornet* implementation for GPU architectures, an experimental analysis, and its comparison with the state of the art dynamic approaches.

The paper is organized as follows. Section II presents an analysis of the state of the art in terms of static and dynamic sparse formats. Section III presents the *Hornet* data structure and its implementation for GPUs. Section IV presents the experimental setup and a detailed empirical analysis. Section V is devoted to the concluding remarks.

II. Related Work

Many linear systems and graph problems arising from the discretization of real-world problems show high sparsity. For many GPU applications, CSR is the de-facto graph representation. Alternative *static* sparse formats include matrix representation for graphs, *COO*, and *Ellpack*.

While some static data solutions allow for dynamic updating, they can only: 1) support a limited number of updates, 2) have a large update time, or 3) have an unacceptable overhead due to data structure re-allocation and re-initialization.

In order to fully support *dynamic* graph algorithms, more advanced and complex data structures have been recently proposed. Their goal is to *efficiently* support dynamic operations in graphs or matrices like edge/node insertions, deletions, or value/weight updates.

The *STINGER* data structure [1] was first introduced as a dynamic graph structure for both temporal and spatial graphs with meta-data for multi-core architectures.

cuSTINGER [2] extends the *STINGER* data-structure to the GPUs. While the *STINGER* and *cuSTINGER* support many similar features, their data structures are very different. *STINGER* relies on blocked linked lists, whereas *cuSTINGER* uses arrays for the neighbor lists.

GraphIn [6] and its extension for GPUs, *Evograph* [3], allow for incremental graph processing on CPU-based architectures by combining two static graph data structures: CSR for the original input and a dynamic edge-lists (COO) to store new edges. These frameworks are constrained to a limited number of updates (pre-defined by the users). COO can lead to scattered memory accesses in case of large updates.

AIM [4] implements a block linked-list data structure for GPUs by using a *STINGER*-like data structure. It allocates a single array and, according to *AIM* [4], it allocates the entire GPU memory for just the graph. By using a single allocation, the initialization is fast and

TABLE I
COMPARISON OF SPARSE GRAPH AND MATRIX REPRESENTATIONS. m_e REPRESENTS THE TOTAL NUMBER OF AVAILABLE/EXTRA EDGES IN THE GRAPH. INSERTIONS AND DELETIONS COMPLEXITY IS PRESENTED FOR SINGLE UPDATES.

Format	Storage	Duplicate checking	Insertion	Deletion	Reset frequency	Memory reclamation	Fixed mem size allocation	Notes
CSR	$n+m$	/	/	/	For every update	No	Yes	
COO	$2\cdot(m+m_e)$	Enabled Disabled	$\mathcal{O}(m)$ $\mathcal{O}(1)$	$\mathcal{O}(m)$ $\mathcal{O}(m)$	After m_e updates	No	Yes	Poor locality
Evograph [3] **CSR+COO**	$n+m+2\cdot m_e$	Not supported	$\mathcal{O}(1)$	Not supported	After m_e updates or single deletion	No	Yes	Reduced locality. Complex API.
DCSR [5]	$2K*n+m+m_e$	Not supported	$\mathcal{O}(1)+\mathcal{O}(m)$	Not supported	After K batches or m_e edges	No	Yes	Complex API
AIM [4]	whole available GPU memory	Always enabled	$\mathcal{O}(deg_{max})$	$\mathcal{O}(deg_{max})$	Whenever exceed allocated memory	No	Yes	
cuSTINGER [2]	$\mathcal{O}(n+2m+m_e)$	Always enabled	$\mathcal{O}(deg_{max})$	$\mathcal{O}(deg_{max})$	No	No	No	
Hornet	$\mathcal{O}(n+2m)$	Enabled Disabled	$\mathcal{O}(1)^1$ $\mathcal{O}(1)^1$	$\mathcal{O}(1)^1$ $\mathcal{O}(1)^1$	No	Yes	No	

update rate is high. However, such an allocation strategy strongly limits the implementation of any advanced analytic computation as the memory is entirely utilized.

Dynamic CSR (DCSR) [5] is a CSR variant for supporting dynamic updates. When initialized, DCSR is nearly equivalent to the CSR representation. Any update to the graph requires a *concatenation* to the initial CSR data structure. Concatenations involve significant memory overhead, require knowing the number of updates a priori, and require a reorganization after each update.

Table I summarizes and compares the characteristics of the data structures at the state of the art. Section 4 presents the analysis of these data structures empirically.

III. THE *Hornet* DATA STRUCTURE

The *Hornet* data structure has been designed to fully support both *dynamic* graphs and matrices [1]. Fig. 1 gives an overview of *Hornet*, which consists of two tiers: The user interface and the internal representation that is abstracted from the user.

From the user's perspective, each vertex is associated with two main fields: The *number of current neighbors* (i.e., *Used* in the figure) that represents the adjacency list size, and a *pointer* to a dedicated adjacency list. Instead of using standard memory allocation function calls for each adjacency list, which would be extremely inefficient. *Hornet* implements this operation by using three components, which are managed by the internal data manager: *1) block-arrays* for storing multiple adjacency lists, *2)* a *vectorized bit tree* for efficiently finding and reclaiming empty memory *blocks* for the adjacency lists, and *3)* B^+*trees* to manage the *block-arrays*.

A. Block-arrays

Hornet represents the graph through a hierarchical data structure, which consists of an adjacency lists, *blocks*, and *block-arrays*. A *block-array* is an array of equally-sized memory chunks, called *blocks*. Each *block* contains a number of adjacency lists equal to a power of two (we refer to this number as the *bsize*). Block sizes are 2^{bsize} edges. The bsize for each vertex, v, is determined as follows $bsize(v) = 2^{\lceil log_2(deg(v)) \rceil}$. As such, $bsize(v)$ is the smallest power of two that fully contains the block.

[1]Graph based and matrix based problems use different terminology to describe identical concepts. For simplicity, we adopt the graph terminology.

Fig. 1(a) shows, as an example, the *Hornet* layout of the initial graph, which consists of four *block-arrays*: $BA_{0,1}$ (*bsize=1*) has one adjacency list; $BA_{1,1}$ and $BA_{1,2}$ (*bsize=2*) contain four and one adjacency list, respectively; $BA_{2,1}$ (*bsize=4*) contains one adjacency list.

Fig. 1(b) shows the *Hornet* layout after the insertion of the three new edges (the details of the insertion process is discussed in Section III-E). The insertion of edge $1 \rightarrow 7$ requires increasing the size of the adjacency list for vertex 1 as it cannot store additional edges *block* in $BA_{1,1}$. Consequently, Hornet allocates a new *block-array* for blocks of $bsize = 4$ ($BA_{2,2}$) and moves the whole *block* containing the adjacency list in it.

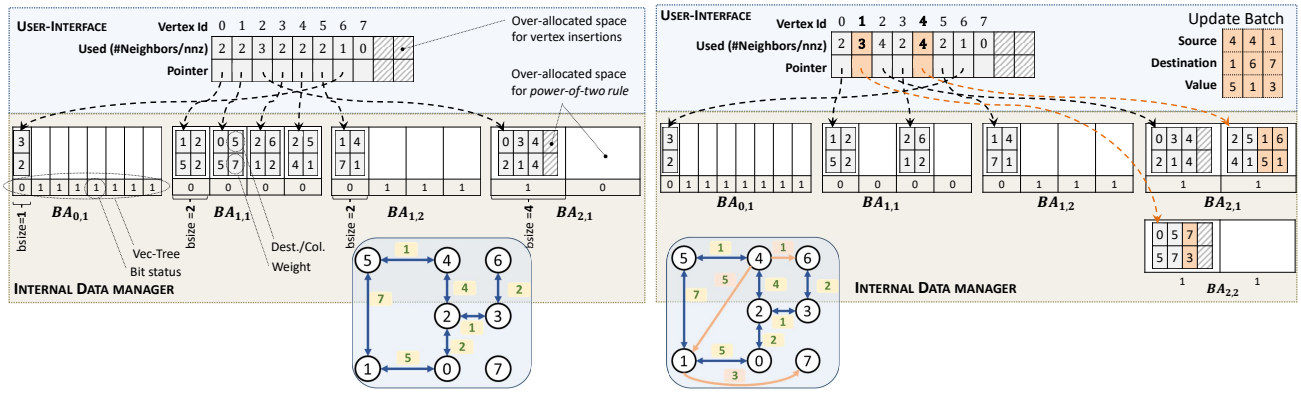
By placing adjacency lists in block sizes that are powers of two, we can place an upper bound on the amount of space allocated for each adjacency list. This allows identifying the worst case upper bound of memory allocated for the entire graph evolution: $2\cdot|E|$. In practice, the average memory allocated for the graph edges, as shown in Section IV, is close to $1.4\cdot|E|$. The number of *blocks* in a *block-array* is also a power of two (as explained in Section III-C).

B. Vectorized Bit Tree

Block-arrays may have empty *blocks* (white spaces in Fig. 1(a), (b)). The *vectorized bit tree* data structure (*Vec-Tree* in the following) is used to efficiently find such empty blocks for new allocations. The Vec-Tree fulfills three key requirements: *1)* to ensure that a new block-array is not allocated until all block-arrays for a given block size are fully utilized, *2)* to have a small memory footprint that does not add significant overhead, and *3)* to find and reclaim empty blocks in an efficient manner.

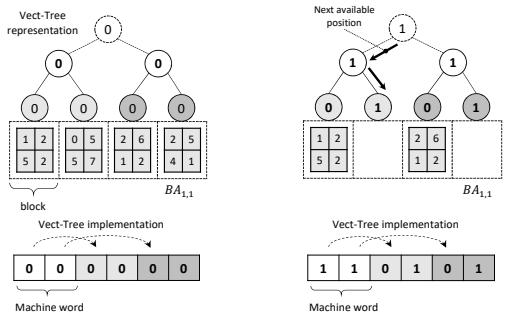
Hornet satisfies the first requirement by associating one Vec-Tree per block-array. Each Vec-Tree consists of a tree of boolean values in which each tree node stores the value of the *logic OR* of its two children. The leafs of the tree represent the state of the *blocks* (1 if empty, 0 if used). Fig. 1 shows the Vec-Trees of all *block-arrays* before and after the graph update. Fig. 2 shows in details the representation and actual implementation of the Vec-Tree of $BA_{1,1}$ before and after the update.

In general, it is possible to see if a *block-array* has an empty *block* by simply inspecting the root. Finding the actual free block can be done within $\mathcal{O}(log(|BA|))$ steps. The same time is spent for an empty block reclamation. Assuming block i as the block of interest, the address of block i is calculated as follows $address(i) = address(BA_{k,id}) + i\cdot 2^k$, where 2^k is the size of each block.



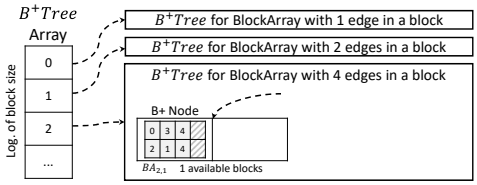
(a) The initial graph. (b) The updated graph.

Fig. 1. *Hornet* layout.

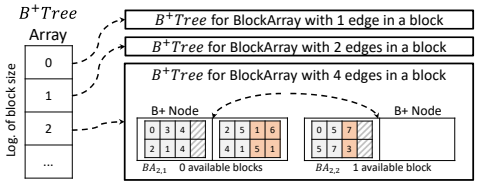


(a) (b)

Fig. 2. Vectorized Bit Tree of *block-array* $BA_{1,1}$. The figure shows the data structure before (a) and after (b) the batch update.



(a) Layout of the initial graph (see Fig. 1(a)).



(b) Layout of the updated graph (see Fig. 1(b)).

Fig. 3. Block-array manager. Specifically, *blocks* of size 4 are emphasized here (also referred to as $BA_{2,i}$).

The simplicity of the lookup enables finding empty blocks at high rates (in contrast to the cuSTINGER implementation that requires a computationally intensive search for finding empty blocks).

C. B^+ Trees of block-arrays

The Vec-Tree layer allows efficiently reclaiming empty blocks for a specific block-array. We adopt a different data structure, i.e., B$^+$Tree, to find a block-array with empty blocks or multiple block-arrays. Even though other data structures, such as linked lists, could be adopted for such a task, *Hornet* implements B$^+$Tree to ensure scalability and efficiency.

Hornet allocates an array of B$^+$Trees, where each B$^+$Tree (one per block size) manages all the block-arrays of a given block size. Fig. 3 shows the B$^+$Tree array for the example of Fig. 1 (initial and after the update), which consists of three B$^+$Tree for blocks of size 1, 2, and 4.

Each node of a B$^+$Tree is a tuple `<data, key>`. The `data` field points to the block-array and the `key` stores the number of free blocks within that block-array. Searching for empty blocks in a B$^+$Tree takes logarithmic time with respect to the tree size. Considering that the size of the block-array is generally big (see Section III-A), the number of block-arrays (the number of nodes in a B$^+$Tree) of a given block size is relatively small. This means that the lookup operations are extremely fast.

As a consequence, when a new block is needed, rather than iterating through all the block-arrays and their corresponding Vec-Trees, all that is needed is query the B$^+$Tree. Each block size with the respective block-arrays are managed by a single B$^+$Tree. Several highly optimized B$^+$Trees implementations already exist in literature (e.g., [7], [8]).

D. Data structure initialization

Hornet allows for graph initialization by starting from an empty data structure and by adding edges and vertices one at a time. It also supports the initialization by starting from a CSR representation and by converting such a static format into the *dynamic-ready Hornet* format.

The data structure initialization consists of three steps. First, for all vertices in the graph, an empty *block* is found based on the degree of each vertex. In the second phase, for performance reasons, all the adjacency lists are temporarily stored in block-arrays and maintained in the host-side rather than being directly copied to the device memory. After that, all block-arrays are copied to the device. Copying the whole block-array instead of single blocks greatly improves the initialization time, since it avoids many small memory transfers while maximizing the PCI-Express bandwidth. Lastly, in the third phase, the

Algorithm 1 Pseudo-code for updating the data-structure after a batch of updates. The pseudo-code for deletions is almost identical to the insertion code by replacing line 4-5.

```
1:  Q ← empty queue                                   ▷ Q : ⟨old_ptr, new_ptr, size⟩
2:  B̂ ← CSR representation of B                        ▷ require sorting: O(B · log(V))
3:  parallel for v ∈ B̂ do                                                  ▷ O(B)
4:      new_degree ← hgraph[v].used + deg_CSR(v)
5:      if (new_degree > BSIZE(hgraph[v].used) then
6:          new_ptr ← MemManager.GETEMPTYBLOCK(new_degree)
7:          ENQUEUE (⟨hgraph[v].pointer, new_ptr, hgraph.used[v]⟩, Q)
8:          hgraph.used[v]     ← new_degree
9:          hgraph.pointer[v] ← new_ptr
10:
                          ▷ Load-balancing is required for efficient copies[2].
11: parallel for q ∈ Q do          ▷ COPYADJCENCYLIST(SRC, DEST, SIZE)
12:     COPYADJCENCYLIST(q.old_ptr, q.new_ptr, q.size)
13: for q ∈ Q do                                                           ▷ O(B)
14:     MemManager.RECLAIMOLDBLOCK(q.old_ptr)
15: parallel for v ∈ B̂ do          ▷ Only for batch insertion              ▷ O(B)
16:     COPYADJCENCYLIST(v.ptr, hgraph[v].pointer, deg_CSR(v))
```

Deletion:
4: new_degree ← *hgraph*[v].used - v.degree
5: **if** (new_degree < BSIZE(*hgraph*[v].used) / 2 **then**

vertex data (degree and adjacency list pointers) are copied to the device.

E. Dynamic Updates

Hornet supports different graph updates: (a) insertion and deletion of vertices, (b) insertion and deletion of edges, and (c) update of values of existing vertices and edges. The first two types (a, b) change the graph topology, while the later changes the data values of the network. Vertex insertions and deletions are implemented through series of edge insertions and deletions, respectively.

Hornet supports graph updates through *batches* [1]–[6], by which different updates are grouped together to maximize system throughput and to avoid sequential latencies.

Algorithm 1 shows the pseudo-code for completing an *edge insertions*. The insertion of new elements in the structure consists of several important yet *parallel* phases. First, the batch is sorted (by source vertex) to improve locality during the update and counts the number of appearances for each row/source (the batch update is converted to a CSR data structure). Then, the vertices requiring additional storage (e.g., vertices 1 and 4 in Fig. 1(b)) are enumerated and queued. A new block is allocated for each of the queued element ($BA_{2,1}$, $BA_{2,2}$), the contents of the old blocks are copied into the corresponding new blocks in parallel, the old block pointers are reclaimed, and the pointers are updated.

The process for edge deletions is similar, and can be obtained by replacing lines 4 and 5 in Algorithm 1 with the lines placed at the bottom of the pseudo code.

Like other approaches in literature (cuSTINGER and AIM), *Hornet* supports *cross duplicate removal* between a batch update and the target graph. The goal is to ensure that the final graph, after the update process, does not contain duplicate edges, which may lead to wrong results in the computation of important analytics (e.g. triangle counting or betweenness centrality).

Hornet support also cross duplicate removal (i.e., edge duplicates between the graph and the batch). Given a single edge, the basic idea is to span a set of threads equal to the degree of the edge source. Each thread maps to a different element in the adjacency list of the source vertex and checks if an edge already exists *Hornet* also implements the removal of *intra-batch* duplicates (i.e., edge duplicates within a batch) by sorting the edges within a batch. The sorting operation is already applied in Algorithm 1 (`line 2`) and both procedures expose high parallelism and efficiency.

[2]The *Hornet* implementation is based on the binary-search load balancing algorithm [9].

TABLE II
GRAPHS AND MATRICES USED IN THE EXPERIMENTAL RESULTS.

Matrix/Graph	Context (Matrix/Graph)	Symm.	Rows, Vertices (M)	NNZ, Edges (M)	Avg.
dblp-2010	Collaboration (G)	Y	0.03	1.6	5.0
Cantilever	FEM (M)	Y	0.06	4.1	65.2
Protein	Protein (M)	Y	0.03	4.3	120.3
Spheres	FEM (M)	Y	0.08	6.1	73.1
Ship	FEM (M)	Y	0.14	7.6	56.5
Wind	Wind tunnel (M)	Y	0.21	11.6	54.4
in-2004	Web crawl (G)	N	1.38	16.7	12.2
soc-LiveJournal1	Social Network (G)	N	4.85	69.0	14.2
cage15	DNA (G)	N	5.15	99.2	19.2
europe_osm	Road (G)	Y	50.91	108.1	2.1
kron_g500-logn21	Synthetic (G)	Y	2.1	182.1	86.8
indochina-2004	Web crawl (G)	N	7.41	194.1	26.2
uk-2002	Web crawl (G)	N	18.5	298.1	16.1
com-livejournal	Ground-truth comm. (G)	Y	4.00	69.3	17.3
com-orkut	Ground-truth comm. (G)	Y	3.07	234.3	76.2

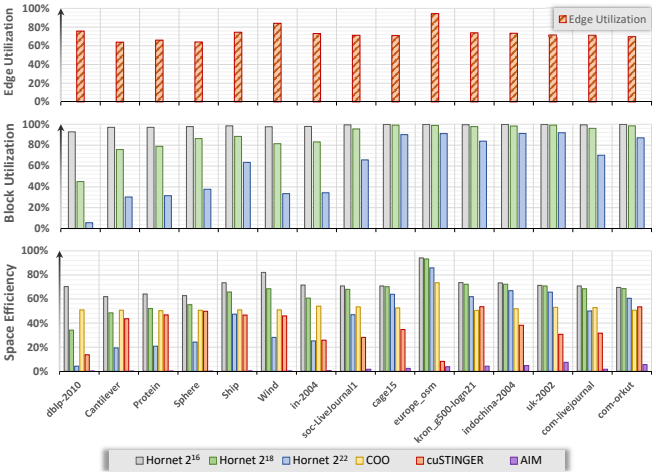


Fig. 4. Upper-side: block-level fragmentation analysis. Middle-side: Fragmentation analysis at block-array level. Bottom-side: Overall memory utilization efficiency.

IV. Experimental Results

Table II reports the set of sparse graphs and matrices used in the experiments and their main characteristics. They have been taken from the University of Florida Sparse Matrix Collection [10].

We conducted the efficiency analysis and the comparison with the corresponding state-of-the-art data structures for GPUs. We consider two key factors for the evaluation, which include *memory utilization* and *update rates*.

The experimental analysis has been conducted on a NVIDIA Tesla (PCI-E) P100 device (Pascal micro-architecture) with Xeon E5-2650 v4 host processor. The P100 consists of 56 SMs with a total of 3,840 CUDA cores and 16GB DRAM memory.

A. Memory utilization efficiency

The *Hornet* memory utilization is evaluated and compared with static data structures (CSR and COO) and dynamic data structures (cuSTINGER, AIM).

We first analyze the *block-level* fragmentation, that is, the unused edges within the blocks due to the power-of-two block sizing. Fig. 4 (upper subplot) reports the results, in which 100% represents no fragmentation (i.e., the entire power of two block is utilized). The bar value represents the average memory utilization in the allocated blocks (e.g., 78% for *dblp-2010*), while the difference (22% for *dblp-2010*) represents the over-allocated memory.

We then analyze the fragmentation at the *block-array level*, that is, the unused and over-allocated memory within block-arrays due to empty blocks (see Section III-B). Fig. 4 (middle subplot) reports the results, by considering different *block-array* sizes: $2^{16}, 2^{19}$, and 2^{22} edges. As expected, as the block-array size increases so does fragmentation (i.e., the storage utilization decreases). This is especially evident for smaller graphs. On the other hand, as will be shown in Section IV-B, larger block-array sizes also have a higher update rate.

Finally, Fig. 4 (bottom subplots) shows the comparison between *Hornet* and the other data structures in terms of overall memory utilization efficiency. *CSR* is chosen as reference point, since it is the most compact state of the art data structure. *CSR* is represented by 100% utilization in the figure. The overall comparison of Fig. 4 underlines that *Hornet* strongly improves (almost twice) the memory utilization efficiency with respect to the best dynamic counterpart at the state of the art (*cuSTINGER*). It also shows that, if properly configured, *Hornet* provides better memory efficiency then the static *COO*The memory utilization of AIM is extremely low due to to the fact that AIM always allocates the entire GPU memory.

B. Update rates

We evaluate the update rates (expressed as updates per second) the dynamic data structure can handle for batches from 1 to 10^7 updates per batch. Similar to cuSTINGER, STINGER, and AIM, *Hornet* verifies that all new edges do not exist in the graph prior to insertion. Other data structures, including EvoGraph and GraphIn, do not perform this in their update phase. As such their update phase is potentially shorter. AIM, Evograph, and GraphIn which use static allocation which do not use memory allocations in their update process. However, these other libraries also need to be re-initialized whenever they need more memory than was originally allocated.

Fig. 5 shows the insertion update rate for four different graphs for both cuSTINGER (a) and *Hornet* data structure (b). Fig. 5 (c) summarizes the speedup of the new data structure compared to the cuSTINGER implementation. For small batches, 10^4 edges and smaller, cuSTINGER outperforms *Hornet*. However, for larger batches cuSTINGER has a performance dip due to communication overhead - *Hornet* does not.

To perform a fair comparison of *Hornet* with AIM, we configured *Hornet* to use a *minimal* block size similar to the one found in AIM. In many cases *Hornet* is faster

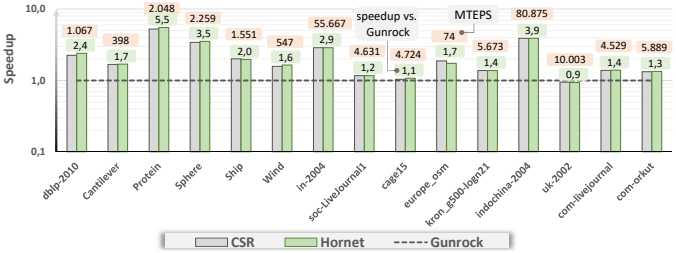


Fig. 7. Performance comparison of BFS between CSR, *Hornet*, and Gunrock (CSR).

than AIM. Fig. 6 depicts the update rate of AIM (a) and *Hornet* (b), and the speedup of *Hornet* compared to AIM (c). Also in this case, *Hornet* shows lower update rates than AIM for small batches, and in particular for graphs with regular degree distribution (*cage15*). On the other hand, *Hornet* outperforms AIM for larger batches up to $82x$ (see Sec. III-E). For the *kron_g500-logn21* graph, *Hornet* is especially faster than AIM as it stores the entire adjacency array in a single block rather than the multiple blocks used by AIM to improve locality. Note that Hornet outperforms AIM even though AIM does not require memory allocations as part of its update process.

The reduced performance of *Hornet* for small batches is in part due to the preprocessing phase that converts the batch update to CSR. This conversion (applied to all batch sizes) is relatively costly for small batches where there is little work to update the graph. However, it greatly improves the performance for large batches.

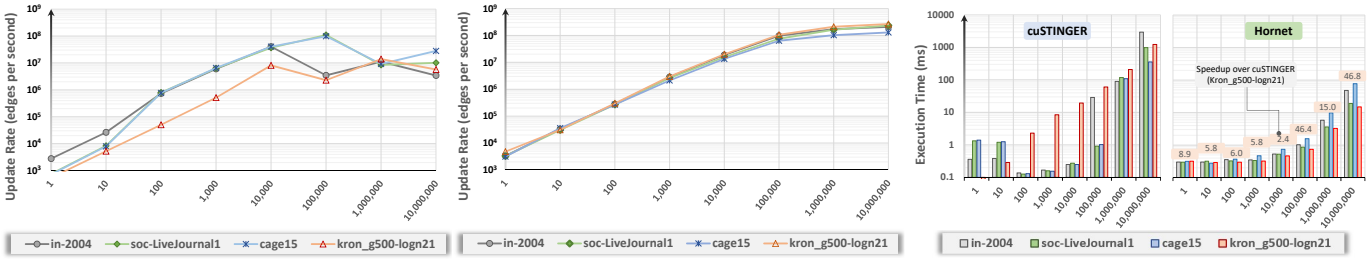
Using the AIM configuration, *Hornet* can process up to 800 millions updates per second (Fig. 6(b)). This can be further increased to *1 billion updates* per second if the duplicate testing is disable as was done in GraphIn and Evograph.

C. Breadth-first search and SpMV

BFS is a fundamental graph operation and building block for most graph algorithms. We compare the performance of *Hornet* and CSR data structures for the BFS graph traversal. In addition, we evaluate our solution with the state-of-the-art CSR implementation provided by the Gunrock library [11]. Fig. 7 shows the speedup and the performance (millions traversed edges per seconds, MTEPS) of our BFS algorithm using CSR and *Hornet* in comparison to the Gunrock implementation[3]. Our implementations were typically faster than Gunrock, in some cases as much as $5.5x$ faster. *Hornet* shows slightly better performance than CSR (up to 10%) thanks to better locality of vertices with the similar degree vertices within the same block-array.

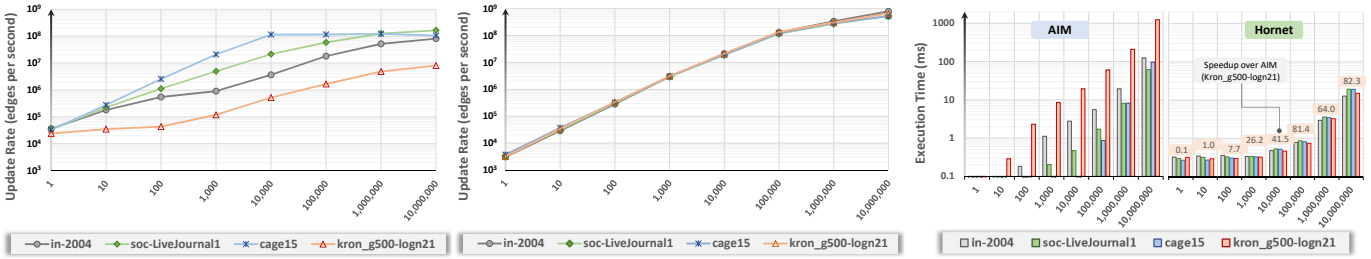
Sparse matrix-vector multiplication (SpMV) is a core primitive in linear algebra and widely used in numerous real-world applications. We evaluate the performance of *Hornet* for SpMV in contrast to CSR and DCSR [5] implementations. Fig. 8 compares the performance of SpMV for these three data representations. For DCSR, we used the implementation provided in [5]. The CSR and *Hornet*

[3]To perform a fair comparison, we evaluate all implementations by forcing traversing exactly the same number of edges (`atomicCAS` and without the idempotent status lookup).



(a) Update rate for cuSTINGER [2]. (b) Update rate for Hornet. (c) Execution time and speedup of Hornet and cuSTINGER [2].

Fig. 5. Analysis of update rate of Hornet against cuSTINGER. Hornet is configured in an equivalent manner to cuSTINGER (minimum edges per *block* = 8, and *block-array* size = 2^{21}).



(a) Update rate for AIM [4]. (b) Update rate for Hornet. (c) Execution time and speedup of Hornet and AIM [4]

Fig. 6. Analysis of update rate of Hornet against AIM. Hornet is configured in an equivalent manner to AIM to ensure the same interaction with the memory manager and avoid new memory allocations (minimum edges per *block* = 256, and *block-array* size = 2^{22}).

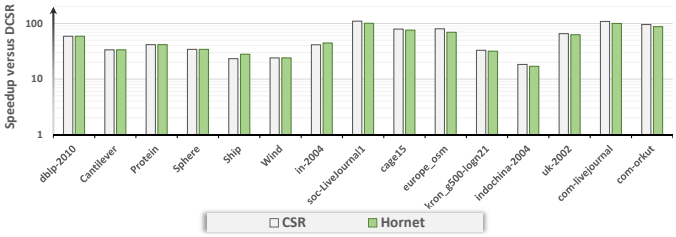


Fig. 8. Performance comparison of SpMV between CSR, DCSR, and *Hornet*. The figure depicts the normalized speedup over DCSR.

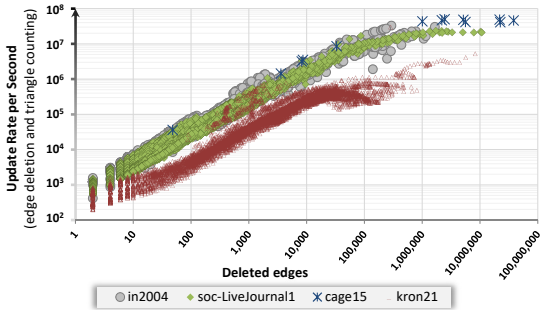


Fig. 9. Update rate per second for finding the maximal k-truss in a graph with *Hornet*.

SpMV implementations are identical except for the data structures used.

Fig. 8 depicts the speedup of the CSR and *Hornet* in comparison to DCSR which has a custom SpMV implementation. We note that *Hornet* is at least $10x$ faster than DCSR and in some cases as much as $100x$ faster.

D. K-Truss

We evaluate the performance of *Hornet* when used to implement a dynamic graph algorithm. We implemented the algorithm for finding the maximal k-truss in a graph presented in [12]. The process of finding k-truss is well known [13] and involves pruning (deleting) edges out of the graph that do not meet an iterative requirement, namely the number of triangles per edge. The exact way that the edges were selected goes beyond the scope of this work.

Fig. 9 depicts the results in terms of update rate per second during the whole algorithm, where each update includes the time spent for a step of edge deletion and the time required for running the dynamic triangle counting.

The results show that, by adopting *Hornet*, the dynamic algorithm is able, at peak rates, to update the graph and run analytics at a rate of roughly 48M updates per second.

V. Conclusions

In this work, we presented *Hornet*, a new GPU data structure for representing dynamic sparse graphs and matrices. *Hornet* supports both insertions, deletions, and value updates. Unlike past attempts at designing dynamic graph data structures, the proposed solution does not require restarting due to a large number of edge updates. We showed that *Hornet* outperforms state-of-art dynamic graph formats in terms of both performance and memory footprint.

Acknowledgment

This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

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

[2] O. Green and D. Bader, "cuSTINGER: Supporting Dynamic Graph Algorithms for GPUS," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2016.

[3] D. Sengupta and S. L. Song, "Evograph: On-the-fly efficient mining of evolving graphs on gpu," in *International Supercomputing Conference*. Springer, 2017, pp. 97–119.

[4] M. Winter, R. Zayer, and M. Steinberger, "Autonomous, independent management of dynamic graphs on gpus," in *International Supercomputing Conference*. Springer, 2017, pp. 97–119.

[5] J. King, T. Gilray, R. M. Kirby, and M. Might, "Dynamic Sparse-Matrix Allocation on GPUs," in *International Conference on High Performance Computing*. Springer, 2016, pp. 61–80.

[6] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, "GraphIn: An Online High Performance Incremental Graph Processing Framework," in *European Conference on Parallel Processing*. Springer, 2016, pp. 319–333.

[7] T. Bingmann, "STX B+ Tree C++ Template Classes."

[8] J. Jannink, "Implementing deletion in B+-trees," *ACM Sigmod Record*, vol. 24, no. 1, pp. 33–38, 1995.

[9] F. Busato and N. Bombieri, "A dynamic approach for workload partitioning on gpu architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1535–1549, 2017.

[10] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

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

[12] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan, V. Prasanna, and D. Bader, "Quickly Finding a Truss in a Haystack," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2017.

[13] J. Cohen, "Trusses: Cohesive Subgraphs for Social Network Analysis," *National Security Agency Technical Report*, p. 16, 2008.