International Journal of Parallel Programming
https://doi.org/10.1007/s10766-024-00774-z

# `Meerkat`: A Framework for Dynamic Graph Algorithms on GPUs

**Kevin Jude Concessao[1] · Unnikrishnan Cheramangalath[1] · Ricky Dev[2] · Rupesh Nasre[2]**

Received: 16 February 2024 / Accepted: 24 July 2024

**Abstract**
Graph algorithms are challenging to implement due to their varying topology and irregular access patterns. Real-world graphs are dynamic in nature and routinely undergo edge and vertex additions, as well as, deletions. Typical examples of dynamic graphs are social networks, collaboration networks, and road networks. Applying static algorithms repeatedly on dynamic graphs is inefficient. Further, due to the rapid growth of unstructured and semi-structured data, graph algorithms demand efficient parallel processing. Unfortunately, we know only a little about how to efficiently process dynamic graphs on massively parallel architectures such as GPUs. Existing approaches to represent and process dynamic graphs are either not general or are inefficient. In this work, we propose a graph library for dynamic graph algorithms over a GPU-tailored graph representation and exploits the *warp-cooperative work-sharing execution model*. The library, named `Meerkat`, builds upon a recently proposed dynamic graph representation on GPUs. This representation exploits a hashtable-based mechanism to store a vertex's neighborhood. `Meerkat` also enables fast iteration through a group of vertices, a pattern common and crucial for achieving performance in graph applications. Our framework supports dynamic edge additions and edge deletions, along with their batched versions. Based on the efficient iterative patterns encoded in `Meerkat`, we implement dynamic versions of popular graph algorithms such as breadth-first search, single-source shortest paths, triangle counting, PageRank, and weakly con-

✉ Kevin Jude Concessao
112014001@smail.iitpkd.ac.in

Unnikrishnan Cheramangalath
unnikrishnan@iitpkd.ac.in

Ricky Dev
cs17b111@cse.iitm.ac.in

Rupesh Nasre
rupesh@cse.iitm.ac.in

[1] Department of Computer Science and Engineering, Indian Institute of Technology Palakkad, Palakkad, India

[2] Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India

nected components. We evaluated our implementations over the ones in other publicly available dynamic graph data structures and frameworks: *GPMA*, *Hornet*, and *faimGraph*. Using a variety of real-world graphs, we observe that `Meerkat` significantly improves the efficiency of the underlying dynamic graph algorithm, outperforming these frameworks.

**Keywords** Dynamic graph algorithms · GPU programming · Parallel computing · CUDA

## 1 Introduction

Real-world graphs undergo structural changes: nodes and edges get deleted, and new nodes and edges are added. Handling dynamic updates poses new challenges compared to a static graph algorithm. Efficient handling of these dynamic changes necessitates (i) how to represent a dynamically changing graph, (ii) how to update only the relevant part of the graph depending upon the underlying algorithm, and (iii) how to map this update effectively on the underlying hardware. These issues exacerbate on massively parallel hardware such as GPUs due to SIMD-style execution, the need to exploit on-chip cache for optimal performance, and nuances of the synchronization protocols to deal with hundreds of thousands of threads. Effectively addressing these issues demands new graph representations, binding of the theoretical and systemic graph processing, and tuning the implementation in a GPU-centric manner. Former research has invented multiple graph representations in diff-CSR [1], SlabGraph [2], faimGraph [3], Hornet [4] and cuStinger [5] to maintain the changing graph structure. The SlabGraph framework [2] proposes the *SlabHash* [6]-based graph data structure and follows warp based execution model, which we build upon in this work.

Dynamic graph algorithms can be categorized as (i) incremental wherein nodes and edges are only added, (ii) decremental wherein nodes and edges are only deleted, and (iii) fully dynamic which involves both the incremental and the decremental updates.

Existing solutions to deal with dynamic graphs are plagued with one of the two issues: they apply to certain types of graphs, or they are inefficient at scale. Thus, the solutions may work well for low-diameter graphs such as social networks but are expensive on road networks which are characterized by large diameters. Alternatively, the algorithms may work on CPUs, but may not be readily translatable to massive multi-threading on the GPUs. Central to solving these issues lie two fundamental questions related to storage and compute: how to represent a dynamic graph, and how to enumerate through a set of graph elements (such as vertices). Graph representation is crucial because the optimal representation for static processing quickly goes awry with dynamic updates. Thus, due to dynamic edge addition, memory coalescing on GPUs can be adversely affected, resulting in reduced performance. Similarly, two types of iteration patterns are common in graph processing: through all the current graph vertices, and through the latest neighbors of a vertex (which change across updates). Both these operations are so common that we treat them like primitives, whose performance crucially affects that of the underlying dynamic graph algorithm. Note that unlike in the case of a static graph algorithm which may suffer from load imbalance due to different threads working on vertices having differently-sized neighborhoods, the issue of load imbalance is severe in a dynamic graph algorithm, as the load imbalance itself may vary across structural updates, leading to unpredictable performance results. This makes applying optimizations in a blanket manner difficult for dynamic graphs and demands a more careful custom processing. Such customization allows the techniques to apply to different algorithms as well as to different kinds of updates for the same dynamic graph algorithm.

In particular, this paper makes the following contributions:

1. We illustrate mechanisms to represent and manipulate large graphs in GPU memory using a hash-table based data-structure. Our proposed dynamic graph framework, `Meerkat`, makes primitive operations efficient (such as iterating through the current neighbors of a node, iterating through the newly added neighbors of a node, etc.).
2. Using the efficient primitives and warp-cooperative work sharing strategy in `Meerkat`, we demonstrate dynamic versions of popular graph algorithms on GPUs: breadth-first search (BFS), single source shortest path (SSSP), Triangle Counting, PageRank and weakly connected components (WCC). Apart from the common patterns among these algorithms, we highlight their differences and how to efficiently map those for GPU processing.
3. We qualitatively and quantitatively analyze the efficiency of our proposed techniques implemented in `Meerkat` using a suite of large real-world graphs and five dynamic graph algorithms. `Meerkat` eases the programming of dynamic graph algorithms and readily handles both the bulk and the small updates to the underlying graph object. The performance obtained by the dynamic algorithms built on top of `Meerkat`'s primitives is significantly better than their static versions.
4. We evaluated `Meerkat` framework against GPMA, Hornet, FaimGraph, which are popular dynamic graph data structures for GPUs.

The rest of the paper is organized as follows. Section 2 describes the background and Sect. 3 describes the motivation for our work. Section 4 describes our proposal in detail, highlighting the graph representation, and efficient implementation of graph primitives. Based on the primitives, Sect. 5 builds various graph algorithms and explains their efficient execution on GPUs. Section 6 quantitatively evaluates the effectiveness of our proposed dynamic graph processing using a suite of large graphs. Section 7 compares and contrasts against related work. We conclude in Sect. 8.

## 2 Background

Frameworks for dynamic graph algorithms on GPUs are hitherto largely unexplored. Selecting an efficient data structure to represent dynamic graph objects is not intuitive as a graph object undergoes insertion and deletion of edges and vertices over time in presence of several worker threads. Providing an efficient framework with good dynamic graph data structure, auxiliary data structures and constructs for programming parallel dynamic graph algorithms is challenging. GPU follows SIMT architecture with a group of threads called `warp` following the same control path. It also has exposed memory hierarchy. Such peculiarities exacerbate the challenges for dynamic graph algorithms on GPUs.

Our framework `Meerkat` addresses these challenges and provides API for developing efficient processing of graphs under structural modifications. It builds upon SlabHash data structure. Awad et al. [2] propose a dynamic graph data structure (which we shall refer to as SlabGraph) that uses the SlabHash data structure [6] for maintaining the vertex adjacencies. SlabGraph provides efficient ways for inserting and deleting edges in dynamic graph objects. Unlike other dynamic graph data structures, such as Stinger [7], GPMA [8], faimGraph [3] and Hornet [4], SlabGraph relies on Warp Cooperative Work Sharing (WCWS) execution model [6], which is crucial for optimal performance on GPUs. In the warp cooperative work-sharing strategy, each of the 32 threads within a warp has a unique piece of work to process. In our context, each thread is mapped to a unique vertex. All threads of a warp need not have a vertex whose adjacencies must be processed. A warp may have up to 32 vertices to process. These pieces of work are serialized by an intra-warp work queue to be processed *one at a time* by all the warp threads in parallel. Two intra-warp communication operations are crucial: (i) `ballot` polls for threads in a warp which have items to process using a boolean expression. (ii) the `shfl` warp-wide intrinsic broadcasts this item from the elected thread to the entire warp, to be processed by all the warp threads. The warp cooperative work-sharing execution model desires that all the threads of a warp are active at any point for the successful execution of the warp-cooperative intrinsics it relies on. The benefit of this is that it avoids warp divergence, and enables coalesced access of the neighbours of a vertex. We discuss SlabGraph in detail below.

## 2.1 SlabGraph Data Structure

An efficient dynamic graph structure demands a dynamic adjacency list. SlabGraph exploits a concurrent hash table for every vertex, to store adjacency lists using a form of chaining. The data structure is designed and optimized for warp-based execution on the GPU. SlabGraph allocates a SlabHash object for each vertex. A SlabHash object has a fixed number of slab lists (buckets) for a vertex $v \in G.V$ determined a priori, where G is the graph object. The number of slablists allocated for a vertex $v$ is $\lceil \frac{outdegree(v)}{loadfactor \times slabsize} \rceil$, where $0 < loadfactor < 1$ and slabsize is 31 and 15 for unweighted and weighted graphs respectively. Lower values of loadfactor allocate more *slablists* per vertex. Insert, Delete, and Query operations for an edge (*u,v*) use a hash function to index into one of the allocated slab lists. The hash function depends upon both the source vertex *u* and the destination vertex *v*. A *slablist* is a linked list of *slabs* and is also called a *bucket*. Each *slab* is 128 bytes long to match the L1 cache line size for coalesced memory access within a single warp. The adjacent vertices of a source vertex are stored in one of the slab lists determined by a hash function. The 128 bytes in a slab form 32 lanes with 4 bytes per lane (32 is the GPU's warp size). Each lane is to be processed by a corresponding thread in the warp. The last lane is reserved for storing the address of the next slab. SlabHash's ConcurrentSet (Concurrent Map, respectively) is used for unweighted (weighted, respectively) graphs to store the

**Table 1** Summary of elementary components of SLABGRAPH data structure

| Item | Use | Description |
|---|---|---|
| Slab | Stores weighted or unweighted neighbours of a vertex | 128-byte memory block |
| Slablist (Bucket) | Stores adjacency of a vertex | Linked-list of slabs; the adjacencies of a vertex could be distributed over multiple slablists as determined by a hashing function |
| Head slab | – | First slab of a slablist |
| Collision slab | Accommodate vertices that cannot be stored in the current slab list | Slabs chained sequentially from the end of the head slab |
| ConcurrentSet | Store the adjacencies of a vertex using a set of slablists in an unweighted graph; one instance for every vertex | Hashtable; each slab storing up to 31 unweighted adjacent neighbours, for a specific vertex |
| ConcurrentMap | Store the adjacencies of a vertex using a set of slablists in a weighted graph; one instance for every vertex | Hashtable; each slab storing up to 15 pairs of adjacent neighbours and their corresponding weights, for a specific source vertex |
| Device—context object | Maintains pointers to slab lists of a vertex, and book-keeping data structures (such as degree count, slab list size, etc.) | GPU device memory instance of ConcurrentSet / ConcurrentMap for a vertex |

adjacent neighbours for each vertex. Every slab in the CONCURRENTSET can store up to 31 neighbouring vertices ($31\times4$ bytes). Every slab in the CONCURRENT MAP can store up to 15 neighbouring vertices and edge weights ($15 \times 8$ bytes). The last lane in a slab for CONCURRENTSET and CONCURRENTMAP is reserved for storing the address of the next slab. Each SLABHASH object for a vertex maintains one device pointer for each slablist allocated for the vertex using a context object. Table 1 lists elementary components of SLABGRAPH data structure.

Figure 1 shows the SLABGRAPH data structure with the vertex $v_i$ having three slablists. Initially, each slablist is allocated with one empty slab which we call *headslab*. When the head slab becomes full, new slabs are chained to the slablist. In Fig. 1, adjacencies of vertex $v_i$ are stored in three slablists. The slablist $v_i[0]$ has two slabs, slablist $v_i[1]$ has only the headslab and $v_i[2]$ has three slabs.

According to the implementation heuristics of SLABGRAPH, a graph with an average degree greater than 15 would allocate approximately $2\times$ more slabs for the weighted SLABGRAPH representation with CONCURRENTMAP than the unweighted representation with CONCURRENTSET. While all threads participate in retrieving a single CONCURRENTSET slab, only 31 threads participate actively in query/traversal operations, since their corresponding slab lanes potentially could have vertex data. A CONCURRENTMAP slab is used for representing weighted graphs. It can store up to 15 pairs of the neighbouring vertices and their respective edge weights. When a slab is retrieved by a warp, every pair of a neighbouring vertex and an edge weight is fetched by a pair of threads. While 30 threads are involved in fetching edge-related data, only 15 threads process 15 pairs in the CONCURRENTMAP slab. The last thread in

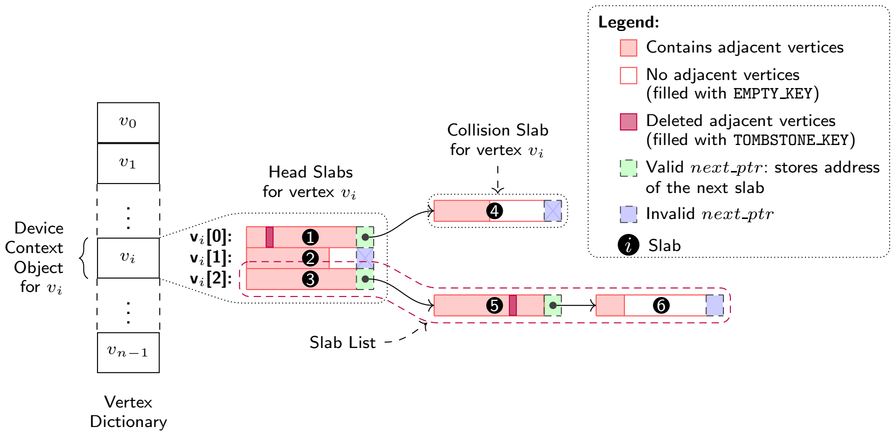

Legend:
Contains adjacent vertices
No adjacent vertices (filled with EMPTY_KEY)
Deleted adjacent vertices (filled with TOMBSTONE_KEY)
Valid next_ptr: stores address of the next slab
Invalid next_ptr
Slab
Collision Slab for vertex $v_i$
Head Slabs for vertex $v_i$
$v_0$
$v_1$
$v_i$
$v_{n-1}$
Device Context Object for $v_i$
$v_i$[0]:
$v_i$[1]:
$v_i$[2]:
Slab List
Vertex Dictionary

**Fig. 1** SlabGraph data structure

ConcurrentMap and ConcurrentSet fetches the next slab's address and is used for performing traversal to the next slab.

An `EMPTY_KEY`[1] is stored in a slab lane if it has not been populated with an adjacent vertex previously, and with a special `TOMBSTONE_KEY`[2] if the slab lane previously held a valid vertex, and is now deleted. Elements within a slab are *unordered*, allowing efficient concurrent access. A slab can be processed efficiently by all the threads of a warp by using warp-wide communication intrinsics such as `__ballot_sync`, `__shfl_sync`, and `__shfl_down_sync` [9]. The warp-cooperative work strategy (WCWS) for searching in a SlabHash hash table is described by [6] and is used in the `Meerkat` framework.

Each SlabHash object for a vertex maintains one device pointer for each slablist allocated for the vertex using a context object. Every graph search operation indexes into an array of SlabHash context objects to retrieve the object for the source vertex of an edge. The particular slablist to store the destination vertex of an edge is determined by using a hash function. The target slablist is then linearly traversed by the warp which has the source vertex in its work queue. For insertions, if the slablist is full, the underlying SlabHash data structure invokes a custom allocator to obtain a new slab that gets added to the end of the slablist. The SlabGraph context object provides APIs for vertex adjacency access and graph manipulation operations inside a device (GPU) kernel by utilizing a work-cooperative work strategy. The SlabHash context object supports methods such as `Insert()` and `Delete()` which execute in a warp-cooperative fashion. These methods are internally used by SlabGraph's device API such as `InsertEdge()` and `DeleteEdge()`, for inserting and removing adjacent vertices for a specific vertex respectively.

[1] `EMPTY_KEY` is defined as `UINT32_MAX-1` for `ConcurrentSet` and `UINT64_MAX-1` for `ConcurrentMap`.

[2] `TOMBSTONE_KEY` is defined as `UINT32_MAX-2` for `ConcurrentSet` and a 64-bit pair ⟨`UINT32_MAX-2, UINT32_MAX-2`⟩ for `ConcurrentMap`.

## 3 Motivation

The SlabGraph data structure suffers from the following shortcomings: (i) it does not provide efficient ways to traverse the current vertex adjacencies, which are crucial for implementing optimized dynamic graph algorithms, (ii) its memory management module is inefficient, and (iii) lack of programming abstractions to ease the programming of dynamic graph algorithms. `Meerkat` framework addresses these shortcomings. `Meerkat` provides optimizations for the dynamic graph data structure, iterators for programming graph algorithms, and warp level API functions such as *reduction*, *broadcast*, and *dequeue*.

When we look into SlabGraph's (the `dynamic-graph` branch of the GunRock) public repository,[3] the source code does not appear to be complete. From the repository point of view: Implementation such as update/delete operations are missing. Many methods (for example, SlabGraph to CSR conversion (and vice versa), the CPU API for batch edge insertion/deletion of edges) appear to be incomplete stubs. It is also not clear how the SlabAlloc allocators are integrated with the underlying SlabHash [6] data structures, for the (de)-allocation of collision slabs.

Our work `Meerkat` builds and improves upon SlabGraph [2] by extending the publicly available source code for `SlabHash`.[4] The `Meerkat` framework extends SlabGraph data structure to a framework for dynamic graph analytics using the WCWS execution model. `Meerkat` also provides Frontier auxiliary data structure to support efficient implementation of dynamic graph algorithms.

Unlike SlabGraph, the `Meerkat` iterators are oblivious to the number of slabs for each vertex. Our iterators provide the ++ operator abstraction to obtain the next slab in sequence. When the available slabs are exhausted, the ++ operator yields a logical sentinel value. In `Meerkat`, we provide *three different iterator abstractions* (see Sect. 4.3) and *two different iteration schemes*, for programming our dynamic graph algorithms. `BucketIterator` provides traversal only over a specific slab-list. The `SlabIterator` internally maintains a `BucketIterator` to traverse over slabs over all the slab lists. The `UpdateIterator` also builds over the `BucketIterator` abstraction with the additional ability to traverse over incrementally updated slabs.

*With the help of cooperative groups* [10], `Meerkat` *uses the shared memory available private to each thread block efficiently.* This is used in the implementation of single source shortest Path (SSSP), breadth-first search (BFS) (Sect. 6.2), and in triangle counting (with sorted adjacencies, see Sect. 6.4).

The limitations with traversal scheme and memory allocation in SlabGraph are described in below subsections.

### 3.1 Traversal in SlabGraph

Traversal over the neighbours of a vertex is achieved with the help of an iterator abstraction. Algorithm 1 shows how neighbours of a vertex $src$ are retrieved, in SlabGraph.

[3] https://github.com/gunrock/gunrock/tree/dynamic-graph.

[4] https://github.com/owensgroup/SlabHash.

Firstly, there is a call to the $size()$ method (in line 1), which iterates over each slablist and counts the number of slabs depending on the number of edges stored in the slab list. This slab count is aggregated over all the slab lists. This operation takes linear time proportional to the number of slabs allocated a priori (a vertex with a large out-degree will have a larger number of initial slab lists allocated for storing its adjacent neighbours). Each slab is identified by a unique index $i$. The base slabs have an index $0 < i < number\,of\,slablists \leq size()$. Accessing the base slabs is fairly simple. However, to access a specific collision slab, say slab 6, stored in slablist 2 (see Fig. 1), the iterator counts the number of collision slabs starting from the first slablist. The iteration stops when the required slablist is found: that is, the number of slabs counted so far up to slablist 2 exceeds the slab index $i$. Further, a traversal is performed within slab list 2 until the slab with the required index $i$ is found. Traversal within a slab list incurs additional memory access with the next pointer $next\_ptr$ of the slab until the required slab $i$ is found.

**Algorithm 1:** SlabGraph - Traversal of a neighbours of a vertex

```
  /* Collision slabs are indexed starting from the first head slab
     */
  /* Convention for slablist:
     $\boxed{\text{Head Slab}} \rightarrow \boxed{\text{Collision Slab}} \rightsquigarrow \cdots \rightsquigarrow \boxed{\text{Collision Slab}}$   */
  /* Example:                                                        */
  /* $SlabList[0] : \boxed{1} \rightarrow \boxed{4}$                 */
  /* $SlabList[1] : \boxed{2}$                                       */
  /* $SlabList[2] : \boxed{3} \rightarrow \boxed{5} \rightarrow \boxed{6}$   */
1 int limit = $iter_{src}$.size() // Count the number of slabs for source vertex $src$
2 for $i \leftarrow 0 \ldots (limit - 1)$ do
      /* Step 1: Retrieve the neighbours in the $i^{th}$ slab          */
3     VertexT dst = $iter_{src}$.value($i, n$) /* dst=neighbour at index $n$ $(0 \leq n \leq 31)$
          in the $i^{th}$ slab                                        */
      /* Step 2: process neighbouring vertex $dst$ here.              */
4 end for
```

*What are the drawbacks of neighbourhood traversal in* SLABGRAPH? For traversal algorithms (such as BFS, SSSP, and PageRank), retrieval of neighbours of vertex (shown in lines 2–3, Algorithm 1) incurs an *overhead from two-pronged linear traversal for each collision slab*: one traversal along the head slabs of the slab lists, and one traversal within the slab list. The running time, is thus, proportional to the slab list count and the (average) slablist length. Indexed access of slabs is not meaningful for the traversal of neighbours in traversal algorithms such as BFS/SSSP/PageRank.

### 3.2 Memory Usage of SLABGRAPH and MEERKAT

The original SLABHASH data structure assumes the responsibility of allocating the head slab (first slab of each slablist) via `cudaMalloc()` function. This is required for maintaining the dynamic graph capabilities of SLABGRAPH. Every vertex has ownership of one SLABHASH object, which contains at least one slab list. The exact

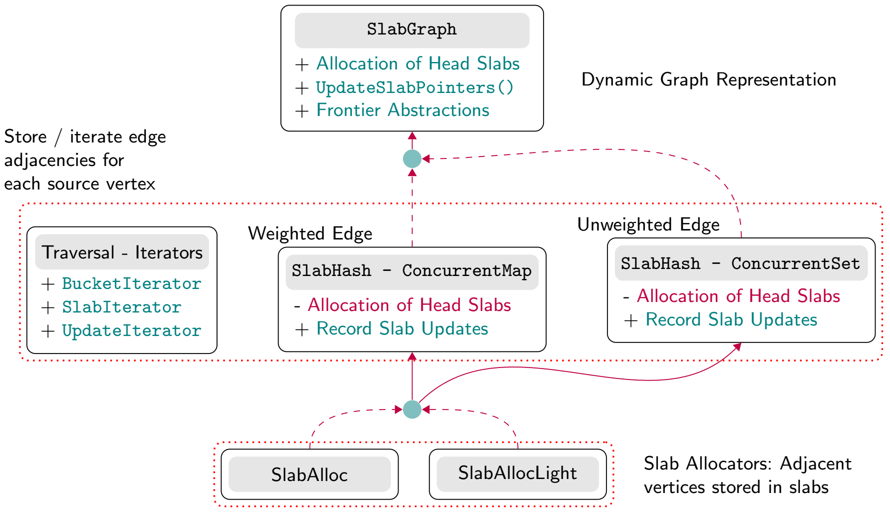


**Fig. 2** Our proposed `Meerkat` framework and its dependencies

number of slab lists received by each vertex is dictated by the initial number of vertices and the reciprocal of the load factor.

Since real-world input graphs often have millions of vertices, we observed that a large number of `cudaMalloc` calls (as many as the number of vertices) for a slab of size 128 bytes results in a significant explosion in the total memory allocated, much beyond the theoretical limit. To alleviate this issue, `Meerkat` makes a crucial design change: it moves the responsibility of allocating the head slabs from SLABHASH to the (outer) SLABGRAPH object which decides the number of slabs required per vertex according to the load factor. A large array of `head_slabs` with the needed space is allocated using a single `cudaMalloc()` function call. Each vertex is assigned a specific number of head slabs according to the initial degree. We maintain an array `slab_list_count` such that `slab_list_count[v]` is the initial number of head-slabs allocated for a given vertex `v`. By performing an `exclusive_scan` operation on the entries of the `slab_list_count` array, we can determine the offset to the head slab for each vertex.

We observe significant savings. As the number of vertices increases, the memory requirement increases rapidly, if the memory allocation for head slabs is performed by SLABHASH objects. It is particularly visible for *LJournal* and *USAfull* input graphs, with the latter going out of memory (`OOM`).

## 4 The `Meerkat` Library

Our work `Meerkat` builds upon and significantly enhances SLABGRAPH [2] by extending the publicly available source code for `SlabHash`.[5] Fig. 2 shows the extensions done by us to SLABGRAPH in our framework `Meerkat`.

[5] https://github.com/owensgroup/SlabHash.

Dynamic graph algorithms on GPUs demand two crucial considerations: (i) memory efficiency due to dynamic updates, and (ii) computation efficiency since the dynamic processing should be faster than rerunning the static algorithm on the modified graph. Based on this goal, `Meerkat` offers a two-pronged approach. In `Meerkat`, we move the responsibility of allocating the head slabs in `SlabHash` outside (to SlabGraph part of `Meerkat`) for all the vertices, as the framework has a better picture of the overall allocation. `Meerkat` efficiently uses the shared memory of GPU (Sect. 4.2) and uses warp cooperative execution model to further optimize the dynamic graph processing.

Second, `Meerkat` provides a set of iterators for traversing through the neighbours of a vertex, which is a fundamental requirement for almost all graph algorithms, such as BFS and SSSP. SlabGraph [2] focuses mainly on the representation and operations of dynamic graphs. In many incremental algorithms such as weakly-connected components, it is sufficient to process the updates performed on the graph representation. Our iterators in `Meerkat` (discussed in Sect. 4.3), enable us to traverse through individual slab lists selectively, through all the slablists for a vertex, or visit only those slabs holding new updates, depending on the requirements of the underlying dynamic graph algorithm. This helps us improve performance.

## 4.1 `Meerkat` API

The `DynamicGraph` data structure of `Meerkat` on *host* and *device* provides API to ease programming dynamic graph algorithms. *Host* API functions `InsertEdges()` and `DeleteEdges()` are provided for inserting and deleting a batch of edges respectively. On the *device* side, *GetEdgeHashCtxts(src)* returns *device-context SlabHash object* for accessing the neighbors of a vertex *src*, through our iterator abstractions. The `Meerkat` framework provides different types of iterators to traverse over the neighbours of a vertex. These iterators are named as `SlabIterator`, `BucketIterator`, and `UpdateIterator` (see Sect. 4.3). These iterators are equipped with functions `begin()`, `end()`, `beginAt()`, and `endAt()`, briefed in Table 2.

`Meerkat` provides API for warp cooperative work sharing execution model. Warp cooperative execution relies on each warp processing the neighbours of the same vertex, using warp intrinsics. The warp maintains a queue of such vertices which are elected in turns, in First in First Out (FIFO) fashion, using lane-id's of the threads in a warp. Meerkat provides an abstraction for such a queue (FIFO) for each warp. This is implemented using the warp level primitives `__ballot_sync()` and `__ffs()`. The pseudocode for the enqueue operation of the warp-private queue is given in Algorithm 2. The Meerkat framework also provides API for warp level reductions and broadcast.

## 4.2 Frontier

A frontier stores the set of graph elements to be processed. A frontier $F$ of type $F{<}T{>}$ is internally an array of elements of type `T`. Each frontier object supports

**Table 2** `Meerkat`: iterator API of `SlabHashCtxt` representing a source vertex

| API | Parameters | Description |
|---|---|---|
| `begin()` | – | Returns a `SlabIterator` to the first slab, in the first slab list for a vertex |
| `end()` | – | Returns a sentinel `SlabIterator` to the logically last slab, in the last slab list for a vertex (implemented using an invalid address) |
| `beginAt()` | `index` | Returns a `BucketIterator` to the first slab in the `index`'th slab list for the source vertex |
| `endAt()` | `index` | Returns a `BucketIterator` to a logically last slab in the `index`'th slab list |
| `updateBegin()` | – | Returns an `UpdateIterator` to the first slab holding incremental updates. The iterator is invalid if updates are not available for a vertex |
| `updateEnd()` | – | Returns an `UpdateIterator` to the logically last slab |

integer-based indexing for accessing its elements by our kernel threads, along with a $size$ attribute. `Meerkat` *privatizes* `Frontier` object into a shared memory partition that is exclusive to a warp. A warp-exclusive partition removes the need for block-level synchronization for overcoming memory hazards. Thus each warp enqueues the frontier edges into its shared memory partition, and flushes them into the global memory frontier on exhaustion. The writes from the shared memory partition to the global memory frontier are performed as a sequence of coalesced writes with the help of a warp-stride loop full memory bandwidth. Further, a private partition ensures that warps can work independently. Insertion of elements into a frontier object is performed by the warp-cooperative `EnqueueFrontier()` function (see Algorithm 2). The `EnqueueFrontier()` function takes a frontier object $F$, an edge $e$ to be enqueued, three array pointers, (namely $src$, $dst$, and $wgt$), for storing the frontier edges to be flushed to the global memory frontier $F$, and $size$ parameter. The $size$ parameter refers to a thread-local variable; each warp thread redundantly storing the number of frontier edges cached within its warp-exclusive shared memory partition at a given point. This redundancy enables every thread determine its offset to enqueue its edge, without relying on intra-warp communication. The number of edges to be enqueued by a warp is calculated and stored in the variable *edge_n* using the functions *__ballot_sync()* and *popc()* (see lines 2–3, Algorithm 2); the shared memory partition is flushed if it cannot accomodate new frontier edges (see lines 4–5). Warp threads holding a valid edge (line 7) compute a unique offset (line 8), and frontier edge is subsequently enqueued (lines 9–11). The thread-local $size$ parameter is updated by the number of edges enqueued by the warp (line 13); the `FlushQueue()` function first increments the size of the global memory frontier $F$ by the number of edges in the shared memory partition (line 18) using a single *atomic* compare-and-swap, and empties the cached enqueued frontier edges into the global memory frontier using a warp-size stride loop (see lines 22–24).

The BFS and SSSP computation in `Meerkat` rely on using a pair of frontiers for driving their iterations: a frontier $f_{current}$ holding a set of edges whose destination

**Algorithm 2:** `Meerkat` Warp device API: Frontier Enqueue

```
device function EnqueueFrontier (Frontier<Edge> F, Edge e, Vertex *src, Vertex
*dst, uint32 *wgt, uint32 &size) {
    uint32 bitset = ballot_sync(e == INVALID_EDGE)
    /* Flush the shared memory partition if more edges cannot be
       inserted by warp                                          */
    uint32 edges_n = popc(bitset)
    if (size + edges_n > SMEM_MAX_QUEUE_LENGTH) then
        FlushQueue(F, src, dst, weight, size)
    end if
    /* Insert a valid edge into the shared memory partition      */
    if ($e \neq$ INVALID_EDGE) then
        uint offset =
        popc(brev(bitset) & (UINT32_MAX << (WARP_SIZE − warp_thread_rank())))
        src[size + offset] = e.src
        dst[size + offset] = e.dst
        wgt[size + offset] = e.wgt
    end if
    size += edges_n // size: thread-local variable received from the
       caller global kernel function
}
device function FlushQueue (Frontier<Edge> F, Vertex *src, Vertex *dst, uint32
*weight, uint32 &size) {
    uint offset = 0
    if (warp_thread_rank() == 0) then
        offset = atomicAdd(&F.size, size)
    end if
    /* Obtain the offset into the frontier to start flushing     */
    warpbroadcast(offset, 0)
    uint32 index = warp_thread_rank()
    /* Warp flushes its shared memory partition into the global
       memory frontier for the next SSSP iteration               */
    while (index < size) do
        F[offset + index] = Edge{src[index], dst[index], weight[index]}
        index += WARP_SIZE
    end while
    size = 0
}
```

vertices must be inspected; the outgoing edges from these destination vertices which have been updated populate the frontier $f_{next}$ to be used for the next iteration.

## 4.3 Graph Primitives

One of the primitive graph operations is to iterate through the neighbours of each vertex.

Our `Meerkat` framework maintains three types of iterators: `SlabIterator`, `BucketIterator`, and `UpdateIterator` (see Table 3). `UpdateIterator` is an optimized version of `SlabIterator` customized for incremental-only graph processing.

**Table 3** `Meerkat` iterators

| Iterator | Description |
|---|---|
| `SlabIterator` | traverses through all the slabs contained in the slablists for a given vertex, one slablist at a time |
| `BucketIterator` | our primitive form of `SlabHash` iterator; traverses through all the slabs of a single slab-list only |
| `UpdateIterator` | traverses through only those slabs containing new adjacent vertices, contained in updated slab-lists |

The unit of access for all `Meerkat`'s iterator variants is a slab. The same API is provided (see iterator-specific methods in Table 4) for both the weighted and unweighted graph representations of `Meerkat`.

A `BucketIterator` is constructed for a specific slablist in the slab-hash table. For example, in Fig. 1, a `BucketIterator` on slablist $v_i[2]$ only can traverse over the following sequence of slabs: ❸ $\rightarrow$ ❺ $\rightarrow$ ❻. Invoking the `begin_at(slab_list_id)` method on a slab hash table, constructs a `BucketIterator` to the first slab of the slab list indexed with `slab_list_id`. The `end_at(slab_list_id)` method returns an iterator for a logical sentinel slab for the slablist indexed with `slab_list_id`.

The `SlabIterator` is used for traversing all the slabs in the hash table of a given vertex. For example, in Fig. 1, a `SlabIterator` for vertex $v_i$ can traverse over the following sequence of slabs: ❶ $\rightarrow$ ❹ $\rightarrow$ ❷ $\rightarrow$ ❸ $\rightarrow$ ❺ $\rightarrow$ ❻. When the first slablist has been traversed, it iterates over the slabs in the second slablist, and so on, until all the slablists for a vertex are visited. The `begin()` method on a slab hash table constructs a `SlabIterator` object pointing to the first slab in the first slablist. The `end()` method returns a `SlabIterator` referring to a logical sentinel slab. See Table 4) for detailed API descriptions.

Our iterators have been designed to be decoupled from the underlying graph representation using ConcurrentSet for unweighted graphs and ConcurrentMap for weighted graphs (See Table 1). Our iterators behave identically in the manner of traversal of slabs and the retrieval of slab content, regardless of whether ConcurrentSet or ConcurrentMap is used for storing the neighbours of a vertex.

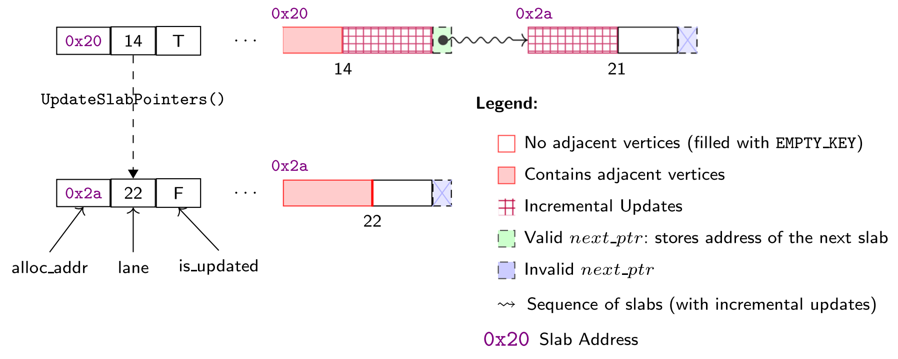
We describe two different schemes for enumerating neighbours. *IterationScheme1* makes use of *SlabIterator*. *IterationScheme2* makes use of *BucketIterator*. Both schemes can apply to all algorithms. *IterationScheme1* and *IterationScheme2* are explained in sufficient detail in Appendix A.1 and Appendix A.2, respectively. Here, we present a summary of the two iteration schemes availabe in `Meerkat`. In *IterationScheme1*, each warp takes at most 32 vertices from the set of active vertices as its work queue. A warp processes its vertices in turns; the neighbours of every vertex (stored in several slabs distributed over multiple slab lists) are accessed with the help of the `SlabIterator` abstraction. The total number of threads spawned by the kernel invocation is equal to the nearest multiple of the warp size above the number of vertices whose adjacencies are to be traversed. In *IterationScheme2*, a warp loops over a queue of slab lists to be processed: each slab list (which holds a partial set of

**Table 4** `Meerkat` primitives

| Function | Description |
|---|---|
| `Meerkat` Iterator-specific methods: | |
| `iter.operator++()` | Advances the iterator to the next slab in sequence |
| `iter.get_pointer()` | Accepts a lane-id (0–31), returns a pointer to an element within a slab with an offset of lane-id |
| `iter.first_lane_id()` | Used when `iter` is an `UpdateIterator`; returns laneid of the first new neighbor in the slab |
| `Meerkat` context object-specific methods: | |
| `G.get_vertex_adjacencies()` | Returns pointer to a device vector of `SlabHash` objects; `i`'th element has neighbors of `i`'th vertex |
| `begin()` | Returns a `SlabIterator` to the first slab, in the first slablist |
| `begin_at(i)` | Returns a `BucketIterator` to the first slab in the `i`'th slablist |
| `update_begin()` | Returns an `UpdateIterator` to the slab at `alloc_addr` located within the first updated slablist |
| `end(), update_end(),end_at(i)` | Returns an iterator to a logically invalid slab (that is, at `INVALID_ADDRESS`) |
| `Meerkat` Other intrinsics: | |
| `thread_id()` | Returns global id of calling thread (`blockDim.x` × `blockIdx.x + threadIdx.x`) |
| `lane_id()` | Returns the position of a thread within a warp (in the range [0, 31]) (`threadIdx.x & 0x1F`) |
| `global_warp_id()` | Returns the global id for a warp (`thread_id() » 5`) |
| `is_valid_vertex()` | Returns `true` if `v` is a valid vertex-id (`v` $\neq$ `INVALID_KEY` and `v` $\neq$ `TOMBSTONE_KEY`) |

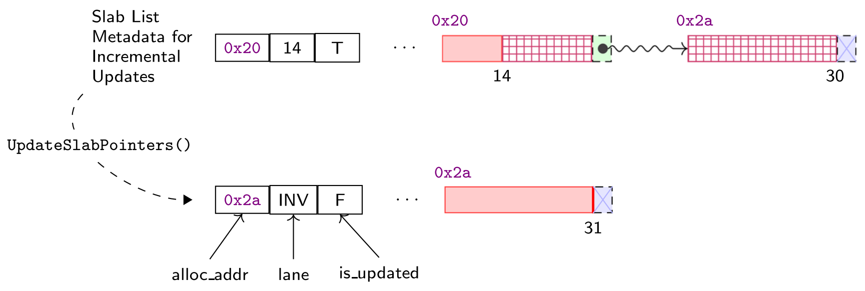
neighbours of the vertex) is identified by a vertex ID and the index of the slab list. If the kernel grid has $n$ warps, every warp processes those slab lists in the queue located at index $k = (warpid + (i \times n)) < queuesize$, where, $i \geq 0$ is a loop variable local to a warp. A slab list is subsequently traversed by a warp using the `BucketIterator` abstraction.

*IterationScheme1* is useful for a majority of situations when the working set of vertices is sufficiently larger than the total number of running warps in the GPU (our GPU has 68 SMs with 64 threads per SM. *IterationScheme1* is beneficial when the working set is larger than 68 SMs × 64 threads per SM = 4352 threads or 136 warps). However, it may not be the case when the working set of vertices is small, and with a possibility that the out-degree of some vertices is unevenly large. It must be recalled that a `SlabIterator` performs a traversal of all slabs, and the distribution of slabs among multiple slab lists has no positive effect in algorithms where all the neighbours of a vertex are to be traversed (that is, all the slabs have to be visited, even though the neighbours are distributed over multiple slab lists). Further, only one warp of threads



0x20
14
T
0x20
0x2a
14
21
UpdateSlabPointers()
Legend:
No adjacent vertices (filled with EMPTY_KEY)
Contains adjacent vertices
Incremental Updates
Valid $next\_ptr$: stores address of the next slab
Invalid $next\_ptr$
Sequence of slabs (with incremental updates)
0x20 Slab Address
0x2a
22
F
0x2a
22
alloc_addr
lane
is_updated

(a) `UpdateSlabPointers()` Case 1: Last slab partially filled with incremental updates



(b) `UpdateSlabPointers()` Case 2: Last slab fully filled with incremental updates

**Fig. 3** Processing incremental updates in `Meerkat`

is responsible for the traversal over all the slabs allocated for a vertex assigned to it. This could lead to a small number of warps ending up performing long traversals over the slabs of a few large-degree vertices, while other small-degree vertices in the same work queue are held hostage for a long time, waiting to be serviced by the same warp.

When the active working set is small, *IterationScheme2* can take advantage of the consequences of hashing, which distributes the neighbours of a vertex evenly over multiple slab lists. *IterationScheme2* avoids the possible tailing effect observed in *IterationScheme1* when the active working set is small, by distributing slab lists of a vertex to different warps for performing the traversal of neighbours. In such situations, *IterationScheme2* enables more warps to access a fewer number of slabs on average and ensures better load balance.

### 4.3.1 `UpdateIterator` for Incremental Graph Processing

In incremental graph algorithms, such as incremental WCC, it is sufficient to iterate over slabs for which new adjacent vertices have been inserted. To facilitate iteration over the updated slabs alone, `Meerkat` maintains the following fields per slablist.

Each slablist is augmented with a `bool` value `is_updated`, which is set to `true` if new edges are inserted into the slablist. Each slablist stores an allocator address field alloc_addr to store the allocator address of the first slab in which new edges have been inserted. Since head slabs are allocated through cudaMalloc(), we use a special value A_INDEX_POINTER in the alloc_addr field, to distinguish the head slab from other slabs returned by the Meerkat allocator. Each slablist also stores the lane-id of the first updated value, in the first updated slab.

Initially, is_updated for a slablist is set to false. The InsertEdge() device method is responsible for setting is_updated for a slablist to true if an insertion occurs at the end of the slablist. For every SlabHash object associated with a vertex, we define UpdateIterators to iterate over only the slabs storing new vertices. In other words, we can traverse only over those slabs in which new vertices have been inserted. An UpdateIterator skips over slablists for whom is_updated is false. Once the updates have been processed, Graph.UpdateSlabPointers() sets the is_updated field to false for all the slablists previously set to true. For such slablists, Graph.UpdateSlabPointers() sets alloc_addr to the last slab in the slablist, and lane id l to the next lane, where subsequent insertions of adjacent vertices are to take place. Figure 3a shows an example portion of a slab list holding the incremental updates. The slab at address 0x20 records the first incremental update at its first free lane holding an EMPTY_KEY (lane 14). This location of the first incremental update in the slablist is recorded by the InsertEdge() device function. Subsequent incremental updates fill up the slab and could be accommodated in more collision slabs chained as a linked list, the last collision slab allocated at address 0x2a (The last collision slab has accommodated incremental updates up to lane 21). Our UpdateIterators rely on this metadata for identifying the incremental updates. Once the incremental updates are processed, they are 'commited' by UpdateSlabPointers() which resets the slab list metadata to the next location where the new batch of incremental updates are to inserted. That is, the address of the last collision slab (0x2a), and its first free lane holding an EMPTY_KEY (lane 22) is recorded in the slablist metadata entries.

In a special case, if the slablist is full (as shown in Fig. 3b), the lane id field lane is assigned a special value INVALID_LANE to denote that the updates would occur at newly allocated collision slabs, chained at the end of the last slab.

In summary, an UpdateIterator behaves like a SlabIterator, but, iterates over slabs that are recognized to be holding incremental updates. Hence, like the SlabIterator, the use of UpdateIterators is compatible with IterationScheme1.

## 5 Dynamic Algorithms using Meerkat

We evaluate Meerkat using the dynamic versions of five fundamental graph algorithms: Weakly Connected Components (WCC), Breadth First Search (BFS), Single Source Shortest Path (SSSP), Triangle Counting (TC), and PageRank (PR). BFS, SSSP, TC, and PR are programmed for both incremental and decremental processing, whereas WCC is programmed only for incremental processing. The BFS, PR, TC, and the WCC algorithm operate on unweighted graphs. On the other hand, the SSSP algorithm requires a weighted graph representation. The fully dynamic versions are implemented with incremental and decremental processing as two computation steps.

## 5.1 Dynamic Single Source Shortest Path and Breadth First Search

The single-source shortest path (SSSP) algorithm, described in Algorithm 3, takes a dynamic graph object $G$, and single source vertex `SRC`, and computes the shortest path to all other vertices from `SRC`. In the dynamic setting, the algorithm is batch-dynamic in nature: it takes a sequence of edge *batches*, where each batch is either an incremental or a decremental batch. The incremental/decremental SSSP algorithm re-computes the shortest paths/distances for the affected vertices in the graph, from the vertex `SRC`. For each node $v$, let $P_v = (SRC \rightsquigarrow \cdots \rightsquigarrow parent(v) \rightarrow v)$ be a shortest path from the source vertex `SRC`. Our SSSP processing is responsible for updating $\langle distance_v, parent(v)\rangle$ pair, where $distance_v$ is the length of the shortest path $P_v$, and $parent(v)$ is the unique predecessor to the vertex $v$ in path $P_v$. Every vertex $v$ other than `SRC` must have a unique $parent(v)$ in a given shortest path $P_v$, which implicitly implies that shortest path of $v$ from the source `SRC` goes through the $parent(v)$. It is therefore understood, that by identifying the $parent(v)$ for every vertex $v$, in its shortest path $P_v$, we are implicitly maintaining a directed tree $T_G$, such that for each edge $e = (u, v) \in T_G$, $u$ is the *parent* of $v$ in $P_v$. Our batch-dynamic incremental/decremental algorithm is responsible for maintaining this *dependence tree*. In the ensuing discussion, a subtree in $T_G$, rooted at vertex $v$, will be represented by $T_v$. A formal discussion on value dependence in shortest distance computation and its representation as a *dependence tree* can be found in [11].

Our static/dynamic SSSP and BFS algorithm implementations on `Meerkat` make use of our frontier based abstractions. The active set of vertices whose distances may be updated, changes from one iteration to the next. The frontier abstraction enables us to visit only this subset of all vertices, avoiding the need for full-graph traversal as needed in iterative SSSP and BFS algorithms.

### 5.1.1 Preliminaries on the Implementation

- *Representation of* TREE- NODE: In our implementation, we have represented the $\langle distance_v, parent(v)\rangle$ pair as a 64-bit unsigned integer, with 32-bits reserved for each half of the pair, where, $distance_v$ occupies the most significant bits, and $parent(v)$ the least significant bits. This allows us to consistently update both halves of a pair with a single 64-bit atomic operation on NVIDIA GPUS.
- *Use of* COOPERATIVE GROUPS: We use cooperative groups for implementing our BFS/SSSP kernels. The kernel grid size is equal to the number of streaming processors (SPs) on the GPU. Every thread block remains resident on the streaming multi-processor (SMs) throughout the lifetime of the kernel, and a grid-stride loop is used for accessing the elements of the *frontier* for each iteration.
- *Use of Shared Memory:* The shared memory of the SMs is equally divided among the warps residing on it. Hence, each warp is assigned its own private region of shared memory, improving access latency.

**Algorithm 3:** Dynamic SSSP - Computation Kernel

```
function SSSP_Dynamic_Kernel (Graph G, Batch batch, tree_node D[],
Frontier<Edge> $F_{current}$, Frontier<Edge> $F_{next}$ ) {
    Vertex_Dictionary *vert_adjs[] = G.get_vertex_adjacencies()
    shared Vertex src[SMEM_MAX_QUEUE_LENGTH]
    shared Vertex dst[SMEM_MAX_QUEUE_LENGTH]
    shared uint32 wgt[SMEM_MAX_QUEUE_LENGTH]
    uint32 size = 0
    uint32 loop_index = thread_id()
    if (batch.is_insertion()) then
        Incremental Algorithm Prologue:
        uint32 loop_bound = roundup(batch.size, WARP_SIZE)
        while ($loop\_index < loop\_bound$) do
            Edge e = INVALID_EDGE
            bool to_consider = false
            if ($index < F_{current}.size$) then
                e = $F_{current}$[index]
                /* Compute and atomically update tree node for vertex
                   e.dst                                                  */
                tree_node $d_{dst}$ = <D[e.src].distance + e.wgt, e.src>
                to_consider = (D[e.src] $\neq$ ⟨INF, INVALID⟩) and (atomicMin(&D[e.dst],
                $d_{dst}$) $> d_{dst}$)
            end if
            SSSP_Frontier_Enqueue(vert_adjs, to_consider, e.dst, $F_{current}$, src, dst,
            wgt, &size)
            loop_index += threads_n() // threads_n() returns number of
               threads in the grid
        end while
    end if
    else
        Decremental Algorithm Prologue:
        Invalidate(batch, D)
        grid sync
        PropagateInvalidation(D, SRC)
        grid sync
        uint32 loop_bound = roundup(vertex_n, WARP_SIZE)
        while ($loop\_index < loop\_bound$) do
            bool to_consider = ($loop\_index < vertex\_n$) and (D[loop_index] $\neq$
            ⟨INF, INVALID⟩)
            SSSP_Frontier_Enqueue(vert_adjs, to_consider, loop_index, $F_{current}$, src,
            dst, wgt, &size)
            loop_index += threads_n() // thread_n() returns total threads
               in the grid
        end while
    end if
    /* Update SSSP distance                                               */
    FlushQueue($F_{current}$, src, dst, wgt, &size)
    grid sync
    SSSP_Frontier_Loop(vert_adjs, $F_{current}$, $F_{next}$, src, dst, wgt, &size, &D)
    grid sync
}
```

It must be recalled that a slab can hold upto 31 adjacent vertices in an unweighted graph, and upto 15 pairs of adjacent vertices and edge weights in a weighted graph.

**Algorithm 4:** SSSP - Frontier Enqueue

```
1  device function SSSP_Frontier_Enqueue (Vertex_Dictionary VA[], bool to_consider,
   Vertex v, Frontier<Edge> $F_{next}$, Vertex *src, Vertex *dst, uint32 *wgt, uint32
   &size) {
2      Edge e = INVALID_EDGE
3      unsigned int32 dequeue_lane = 0
       /* Loop enqueues outgoing neighbours of vertex $v$ */
4      while $((dequeue\_lane = warpdequeue(\&to\_consider)) \neq -1)$ do
5          Vertex current_v = warpbroadcast(v, $dequeue\_lane$)
6          SlabIterator iter = vert_adjs[$current\_v$].begin()
7          SlabIterator last = vert_adjs[$current\_v$].end()
8          while $(iter \neq last)$ do
               /* Iterate over neighbours */
9              Pair p<$dst$, $weight$> = *(iter.get_pointer(lane_id()))
10             Edge $e_{next}$ = INVALID_EDGE
11             if $(is\_valid\_pair(p))$ then
12                 $e_{next}$ = {current_v, p.dst, p.weight}
13             end if
               /* Enqueue outgoing edges of $v$ into next frontier $F_{next}$ */
14             EnqueueFrontier($F_{next}$, $e_{next}$, $src$, $dst$, $wgt$, $\&size$)
15             ++ iter
16         end while
17     end while
18 }
```

Further, the slab occupancy is much lower in low-outdegree graphs such as the road networks. Since the out-neighbours and the respective edge weights are enqueued into the SSSP frontier, low slab occupancy leads to poor utilization of global memory bandwidth. Further, one atomic operation per slab is required to shift the frontier index stored in the global memory. To alleviate this problem, we *privatize the frontier* into a shared memory partition that is exclusive to a warp. A warp-exclusive partition removes the need for block-level synchronization for overcoming memory hazards. Thus each warp enqueues the frontier edges into its shared memory partition, and flushes them into the global memory frontier on exhaustion. Each flush operation involves parallel writes and only one atomic operation to advance the frontier limit. The writes from the shared memory partition to the global memory frontier are performed as a sequence of coalesced writes with the help of a warp-stride loop utilizing full memory bandwidth. Further, a private partition ensures that warps can work independently either in the frontier traversal or in private shared-memory partition flushing mode.

The use of cooperative groups provides for grid-wide synchronization within the kernel, thus avoiding the need for explicitly invoking `cudaDeviceSynchronize()` from the host CPU.

### 5.1.2 Incremental SSSP

The addition of a new edge $(u, v)$ could result in $distance(v)$ only getting reduced if $((distance(u) + weight(u, v)) < (distance(parent(v)) + weight(parent(v), v))$. In such a case, the sub-tree $T_v$ for vertex $v$ is transplanted under a new parent $u$

in $T_G$. All such shortest paths $P_x = (SRC \rightsquigarrow parent(v) \rightarrow v \rightsquigarrow x)$, are now $P_x = (SRC \rightsquigarrow u \rightarrow v \rightsquigarrow x)$. Therefore, it is necessary to re-compute the shortest-path distances for all the vertices in the sub-tree $T_v$ $(v \rightsquigarrow x)$. Our incremental SSSP processing takes an incremental batch of edges as the initial frontier for our static SSSP.

### 5.1.3 Decremental SSSP

If an edge $(u, v)$ which is a part of the shortest path tree $T_G$ is deleted from the graph $G$, then it invalidates $distance(v)$ from the source vertex SRC, and the shortest paths $P_x$ for all vertices $x$ in the subtree $T_v$. If $distance(v)$ is invalidated on the deletion of an edge $(u, v)$, vertex $u$ ceases to be $parent(v)$. This prompts a propagation of invalidations for the shortest distances (from SRC) and the parent vertices determined for all the vertices in $T_v$. In effect, the previously computed $T_v$ ceases to exist in $T_G$. At this juncture, there are three types of vertices in the graph: (i) a set of vertices $V_{valid}$ whose shortest distances and $parent$ information have not been invalidated (ii) a set of vertices $V_{invalid}$ whose shortest distances and $parent$ information have incurred invalidations, as a direct consequence of being destination vertices of deleted edges present in $T_G$, or indirectly, as a consequence of the propagation of invalidation, and (iii) a set of vertices $V_{unreachable}$ which were not part of $T_G$ owing to an absence of a path from the vertex SRC in $G$. Such vertices in $V_{unreachable}$ will continue to remain unreachable even after a batch of edge deletions. Thus, the shortest paths for vertices in the $V_{invalid}$, still reachable from SRC, can be computed by taking all edges $(u, v)$ such that $u \in V_{valid}$ and $v \in V_{invalid}$, as the initial frontier for our static SSSP processing.

We now discuss the specific details of the implementation of the static SSSP, and the incremental/decremental algorithms, presented in Algorithm 3. The static SSSP computation kernel performs frontier-based computation: it accepts a frontier of edges $F_{current}$ and produces a new frontier $F_{next}$ for the next invocation. The SSSP kernel is repeatedly invoked until it produces an empty frontier $F_{next}$. A frontier of type $F{<}T{>}$ is internally an array of elements of type $T$. Each frontier object supports integer-based indexing for accessing its elements by each thread. Every frontier object maintains a $size$ attribute to indicate the number of elements in the frontier array. Insertion of elements into a frontier object is jointly performed by the warp-synchronous `EnqueueFrontier(..)` and `FlushQueue(..)` functions. The tree nodes for all the vertices in the array $D$ (except the source SRC) are initialized with $\langle$INF, INVALID$\rangle$ that is, the shortest path distance for all the vertices is set to INF (infinity), and their $parent(v)$'s to INVALID vertex. The tree node for the source vertex SRC is initialized to $\langle$0,SRC$\rangle$. *We reiterate that the order* $\langle$dist,parent-id$\rangle$ *is important for the SSSP and BFS algorithm implementations.*

The SSSP kernel updates the tree node D[$v_i$]=$\langle d_{current}, p_{current}\rangle$ to $\langle d_{new}, u_i\rangle$ if $(d_{new} = \text{distance}(u_i) + e_i.weight) < d_{current}$. This update is done *atomically* to preserve sequential consistency. The neighbours of $v_i$ are enqueued into the edge frontier $F_{next}$, if $D[v_i]$ is updated.

In Algorithm 3, lines 10–21 define the prologue for the incremental SSSP processing. It takes an incremental batch of edges as the initial frontier $F_{current}$. Lines 14–17 check whether the destination vertex of the edge $e$, that is, $e_{dst}$ has a new shortest path through $e_{src}$. If the shortest path of $e_{dst}$ is updated, the variable *to_consider* is set to `true` (line 17). If *to_consider* is set to `true`, the neighbours of $e_{dst}$ are added to the frontier $F_{current}$ by calling the warp-synchronous $SSSP\_Frontier\_Enqueue(\ldots)$ function (shown in Algorithm 4). This function is based on `IterationScheme1`. A pair of `SlabIterators` are used for traversing over all the slabs holding the neighbours of the vertex $e_{dst}$.

Lines 25–34 define the prologue for the decremental SSSP processing. It involves using the edges in the batch invalidation (see Line 25, expanded in Algorithm 12, in Appendix B) and propagation of shortest path invalidation in the shortest path tree (see Line 27, expanded in Algorithm 13, in Appendix B). Then all the vertices whose distances are valid in the shortest path tree, and their neighbors whose distances are invalid are added to the frontier $F_{current}$ (see lines 29–33). Lines 36–39 define the incremental/decremental computation. The dynamic computation utilizes static SSSP procedure with the frontiers produced by the incremental/decremental prologues for the given batch. The static SSSP computation kernel executes until convergence, i.e., until $F_{next}$ becomes empty. The incremental/decremental BFS processing uses the same kernels as that of incremental/decremental SSSP described in Algorithm 3 (lines 8–34). However, the static algorithm uses a fast *level-based* BFS computation.

We compared the performance of the BFS frontier-based algorithm (on the unweighted graph), and the SSSP algorithm with weights 1 for the edges, with a single slab list allocated for each vertex. The distinction lies in the fact the former can receive up to 31 neighbours per slab access, and the latter receives up to 15 when the slab is fully occupied. Our analysis reveals that when dealing with benchmark graphs where the average degree is approximately 15 and above, the weighted graph representation incurs an overhead of approximately 19.36% (up to 31.7% for Higgs). For graphs with a low degree (average degree 15 or less), the average overhead was under 0.6% (Wiki-talk had an exceptionally higher overhead of 27.7% owing to few high-degree vertices). It is important to note that the neighbours of vertex with an out-degree of 15 or less, can be accomodated within a single slab, regardless of whether the graph representation is weighted and unweighted. Consequently, the difference in the total number of slabs allocated for the entire graph is very small for low-degree graphs. As a result, there is a nearly equal number of memory accesses for slabs, in both weighted and unweighted graphs for the two cases that were discussed here.

## 5.2 PageRank

The PageRank algorithm assigns a score to every vertex in the range [0, 1], which determines its importance in the input graph object. The PageRank value of a vertex in a graph object can be understood as a probability that a random walk in the graph (with $N$ vertices), will arrive at that vertex, computed by an iterative application of equation 1 for all vertices in a sequence of super-steps until a steady state/ convergence condition is met [12].

$$PR(v) = \frac{1-d}{N} + d \cdot \sum_{u \to v} \frac{PR(u)}{|out(u)|} \tag{1}$$

**Algorithm 5:** Page Rank - Static / Incremental / Decremental Algorithm

```
1  function ComputePageRank (Graph G, float PageRanks[vertex_n], float error_margin =
   1e-5, float damping_factor = 0.85, int max_iter = 100) {
2      int iterations = 0
3      float NewPageRanks[vertex_n]
4      float ContributionPerVertex[vertex_n]
5      float delta = 1.0
6      float teleport_value = 0.0
7      while ($delta > error\_margin$ and $iterations < max\_iter$) do
8          FindContributionPerVertex(PageRanks, G.VertexOutDegrees,
           ContributionPerVertex)
9          Compute(G, ContributionPerVertex, damping_factor, NewPageRanks)
10         if (find(G.VertexOutDegrees, 0) == true) then
11             FindTeleportProb(G.VertexOutDegrees, &teleport_value)
12             foreach $i$ in $0 \ldots (vertex\_n - 1)$ parallel do
13                 NewPageRanks[i] += damping_factor $\times$ teleport_value
14             end foreach
15         end if
16         FindDelta(PageRanks, NewPageRanks, &delta)
17         PageRanks $\leftarrow$ copy(NewPageRanks)
18         ++ iterations
19     end while
20 }
```

The pseudocode for static/dynamic PageRank is discussed in Algorithm 5. Algorithm 5 accepts a dynamic graph object $G$, and an array $PageRanks[vertex\_n]$ which identifies the PageRank value for each vertex in the input graph object. In the case of the static algorithm, each element in the array $PageRanks[vertex\_n]$ is initialized with the value $\frac{1}{vertex\_n}$. In the incremental/decremental case, the array element $PageRanks[v]$ contains the PageRank value of a vertex $v$, computed before insertion/deletion. Each iteration of the loop (in lines 7–18) represents a "super-step". The PageRank values of iteration $i$, are determined from those computed in iteration $i-1$. The maximum number of iterations is upper bounded by $max\_iter$. The iterations continue until $delta = \sum_{v \in G.V} |PR_i(v) - PR_{i-1}(v)| > error\_margin$. In other words, $delta$ is the L1- NORM between the PageRank vectors $\mathbf{PR}_i$ and $\mathbf{PR}_{i-1}$, and is computed at line 16. Line 8 initializes $VertexContribution[v] = \frac{PR[v]}{|out(v)|}$ for each vertex $v$, which can be performed with coalesced memory access. The new PageRank values are computed in line 9 according to equation (1) and are adjusted to account for teleportation from zero-outdegree vertices to any other vertex in the input graph object (at lines 10–13). The teleportation probability is added to the PageRank value for every vertex (lines 12–13), if there exists any vertex $v_z$ whose out-degree is zero (see line 10). The teleportation probability to be computed at iteration $i$ is given by $\sum_{v_z} \frac{PR_{i-1}(v_z)}{vertex\_n}$ is computed in the `FindTeleportProb` kernel.

The `Compute` kernel (in Algorithm 6), based on *IteratorScheme1*, describes the computation of PageRank values for all vertices according to equation (1). The `Compute` kernel is invoked with a dynamic graph object $G$ storing incoming edges, an array of PageRank contributions for each vertex, the damping factor, and an array of new PageRank values. Each thread, with thread-id equal to $v$, represents a unique ver-

**Algorithm 6:** Page Rank - Compute Kernel

```
1  function Compute (Graph G, float VertexContribution[], float DampingFactor, float
   NewPageRanks[]) {
2      Vertex_Dictionary *vert_adjs[] = G.get_vertex_adjacencies()
3      if ($(thread\_id() - lane\_id() \ < edge\_n)$) then
4          bool to_compute = (thread_id() < vertex_n)
5          unsigned int pr_value = 0
6          unsigned int32 dequeue_lane = 0
7          while ($((dequeue\_lane = warpdequeue(\&to\_compute)) \neq -1)$ do
8              Vertex current_v = thread_id() - lane_id()+ dequeue_lane
9              SlabIterator iter = vert_adjs[current_v].begin()
10             SlabIterator last = vert_adjs[current_v].end()
11             float local_prsum = 0.0f
12             while ($(iter \neq last)$) do
13                 Vertex u = *(iter.get_pointer($lane\_id$()))
14                 if ($(is\_valid\_vertex(u))$) then
15                     local_prsum += VertexContribution[u]
16                 end if
17                 ++ iter
18             end while
19             warpreduxsum($(\&local\_prsum)$)
20             if ($(thread\_id() == current\_v)$) then
21                 pr_value = local_prsum
22             end if
23         end while
24         if ($(thread\_id() < vertex\_n)$) then
25             NewPageRanks[thread_id()] = pr_value
26         end if
27     end if
28 }
```

tex $v$ in the graph object $G$. Hence, each thread maintains a private variable $pr\_value$ (line 5) to hold the new PageRank value for the vertex it represents. Lines 7–25 compute the new PageRank values for all the vertices collectively represented by the warp, using the warp-cooperative execution strategy with a pair of `SlabIterators`. After selecting a warp lane (in line 7 using the `Meerkat` primitive `warpdequeue()`), line 8 computes the corresponding `id` of the vertex (`current_v`) to be processed by the warp. A pair of `SlabIterators` (lines 9–10) are constructed to traverse the slabs holding the in-edges of vertex $current\_v$ in graph object $G$. The accumulation of the contribution of the in-edges to the PageRank of `current_v` is commutative. Hence, we maintain a thread-local variable $local\_prsum$ (defined at line 11), where we accumulate the PageRank contributions of the neighboring vertices along the incoming edges encountered by the warp threads (line 15). Given that $VertexContribution[u] = \frac{PR[u]}{out[u]}$, re-computing this ratio for every adjacent vertex $u$ of a vertex $v$, leads to two non-coalesced memory accesses for every edge (one memory access for accessing $PR[u]$, and another for $out[u]$) by the warp. Every warp thread uses a neighbour $u$ of the vertex $v$ for indexing into the arrays $PR[]$ and $out[]$. It must be recalled that these neighbouring vertices are fetched by the warp from a slab. Their vertex-id's need not be contiguous as indices, leading to memory accesses by the warp that are non-coalesced). Since these ratios are invariant in every PageRank super-step, they are pre-computed (at line 8, in Algorithm 5), and stored in the array $VertexContribution$, thus reducing the number of non-coalesced memory accesses per edge to one.

**Algorithm 7:** Triangle Counting - Count Kernel

```
1  function Count (Graph $G_1$, Graph $G_2$, Edge edges[edge_n], int *TotalCount) {
2      Vertex_Dictionary *vert_adjs[] = G2.get_vertex_adjacencies()
3      if (($thread\_id() - lane\_id() < edge\_n$)) then
4          bool to_count = (thread_id() < VertexN)
5          unsigned int count = 0
6          unsigned int32 work_queue = 0
7          Vertex u = INVALID_VERTEX
8          Vertex v = INVALID_VERTEX
9          if ($to\_count ==$ true) then
10             u = edges.src[thread_id()]
11             v = edges.dst[thread_id()]
12         end if
13         unsigned int32 dequeue_lane = 0
14         while (($dequeue\_lane = warpdequeue(\&to\_compute)) \neq -1$) do
15             Vertex current_u = warpbroadcast(&u, current_lane)
16             Vertex current_v = warpbroadcast(&v, current_lane)
17             SlabIterator iter = vert_adjs[current_v].begin()
18             SlabIterator last = vert_adjs[current_v].end()
19             while ($iter \neq last$) do
20                 Vertex adj_v = *(iter.get_pointer(lane_id()))
21                 bool edge_present = $G_1$.SearchEdge(is_valid_vertex(adj_v), current_u, adj_v)
22                 if ($edge\_present$) then
23                     count += 1
24                 end if
25                 ++ iter
26             end while
27         end while
28         warpreduxsum(&count)
29         if ($lane\_id() == 0$ and $count \neq 0$) then
30             atomicAdd(TotalCount, count)
31         end if
32     end if
33 }
```

## 5.3 Triangle Counting

Our library's dynamic triangle counting algorithm is adapted from [13], which is based on an inclusion–exclusion formulation. Algorithm 7 consumes a pair of undirected graphs, namely, $G_1$ and $G_2$, and a sequence of $edges$. For each such edge $(u, v) \in edges$, Algorithm 7 computes the cardinality of the intersection of the $adjacency(u)$ in $G_1$ and $adjacency(v)$ in $G_2$ in a warp-cooperative fashion. For each edge, its pair of end-points $u$, $v$ are initialized at lines 10–11. An edge is processed by a warp, one at a time. After electing the thread whose edge needs processing using the `warpdequeue` function of `Meerkat` (line 14), the end-points are broadcasted to the warp threads using warpbroadcast function of `Meerkat` (see lines 15–16). A pair of `SlabIterators` are constructed (lines 17–18) to iterate over the neighbours of vertex $v$ in $G_2$. For each such adjacent vertex $adj\_v$ (line 20), we check if the edge $u \rightarrow adj\_v$ exists. Such an edge indicates the presence of the triangle comprising of vertices $\langle u, adj\_v, v\rangle$, and the thread-local count is incremented by one (see line 23). It must be remembered that each thread in the warp sees a different $adj\_v$; hence lines 21–23 detects different triangles, at the same time. The thread-local triangle `counts` are finally accumulated at warp level using warpreduxsum API of `Meerkat` (see line 28) and then updated to the global variable `TotalCount` (see line 30).

It must be noted that when $G_1 = G_2 = G$ and *edges* is the full set of edges in $G$, Algorithm 7 degenerates to the static triangle counting case. Edge insertions create three types of new triangles: 1) $T_1^i$, triangles with *two old* edges and *one new* edge. 2) $T_2^i$, triangles with *one old* edge and *two new* edges 3) $T_3^i$, triangles with *three new* edges. The undirected nature of the graph also implies that the triangles are computed multiple times. For example, in the static triangle counting case, each vertex of a triangle contributes twice to the triangle count. Hence, the measured count is six times that of the actual count.

**Algorithm 8:** Triangle Counting - Incremental

```
1 function TC_Incremental (Graph PostInsertionGraph, Graph UpdateGraph, Vertex
  Src[vertex_n], Vertex Dst[vertex_n], unsigned int edge_n) {
2     unsigned int $S_1^i = 0$, $S_2^i = 0$, $S_3^i = 0$
3     Count(PostInsertionGraph, PostInsertionGraph, Src, Dst, edge_n, $S_1^i$)
4     Count(PostInsertionGraph, UpdateGraph, Src, Dst, edge_n, $S_2^i$)
5     Count(UpdateGraph, UpdateGraph, Src, Dst, edge_n, $S_3^i$)
6     return $0.5 \times (S_1^i - S_2^i + S_3^i / 3)$
7 }
```

We first find the number of new triangles formed through the intersection of at least one edge. The intersection of the adjacencies of the end-points of the new edges in the post-insertion graph obtains this count $S_1^i$. As such, owing to the undirected nature of the post-insertion graph, computing such an intersection results in a new triangle of type $T_1^i$ being detected twice; a new triangle of type $T_2^i$ is detected four times, and that of type $T_3^i$ is detected six times. Thus, $S_1^i = 2 \cdot T_1^i + 4 \cdot T_2^i + 6 \cdot T_3^i$. This count is obtained in line 3 of Algorithm 8. Next, we detect triangles formed by at least two new edges. Let us call this count $S_2^i$. This is possible if there exists a pair of edges $\langle p, u\rangle$ and $\langle p, v\rangle$ that share a common end-point $p$. Intuitively, at most one old edge pre-existed in the pre-insertion graph, and at least two new edges were added to a common end-point (case 1); or three new edges were added to the pre-insertion graph, (case 2). For each edge $\langle u, v\rangle$ compute the cardinality of the intersections of the adjacencies of $u$ in the post-insertion graph and the adjacencies of $v$ in the update-graph. In case 1, a triangle with two new edges is counted twice. In case 2, a triangle with three new edges is counted six times. Thus $S_2^i = 2 \cdot T_2^i + 6 \cdot T_3^i$. This count is computed in line 4 (in Algorithm 8). Likewise, Intersecting all the edges in the update-graph finds us triangles with only three new edges giving us $S_3^i = 6 \cdot T_3^i$ (See line 5 in Algorithm 8). Thus, in the insertion case, we have $\left|T_1^i\right| + \left|T_2^i\right| + \left|T_3^i\right| = \frac{S_1^i}{2} - \frac{S_2^i}{2} + \frac{S_3^i}{6}$.

**Algorithm 9:** Triangle Counting - Decremental

```
function TC_Decremental (Graph PostDeletionGraph, Graph UpdateGraph, Vertex
Src[vertex_n], Vertex Dst[vertex_n], unsigned int edge_n) {
    unsigned int $S_1^d = 0$, $S_2^d = 0$, $S_3^d = 0$
    Count(PostDeletionGraph, PostDeletionGraph, Src, Dst, edge_n, $S_1^d$)
    Count(PostDeletionGraph, UpdateGraph,Src, Dst, edge_n, $S_2^d$)
    Count(UpdateGraph, UpdateGraph, Src, Dst, edge_n, $S_3^d$)
    return $0.5 \times (S_1^d + S_2^d + S_3^d / 3)$
}
```

Likewise, Algorithm 9 describes the pseudo-code for computing the number of triangles removed after deleting a batch of edges. The number of deleted triangles is given by $\left|T_1^d\right| + \left|T_2^d\right| + \left|T_3^d\right| = \frac{S_1^d}{2} + \frac{S_2^d}{2} + \frac{S_3^d}{6}$.

## 5.4 Incremental WCC

A Weakly Connected Component (WCC) of an undirected graph is a subgraph where all the vertices in the subgraph are reachable from all other vertices in the subgraph. An efficient way to compute the set of all WCCs in a graph object is by using the `Union-Find` data structure. A root-based union-find tree, followed by full path compression can be used efficiently for computing the labels for the vertices, which are representatives of their WCCs, in both the static and the incremental computation.

*Incremental WCC*: find the WCCs, for inserting batches of updates, iteratively. After inserting a batch of edges, we identify the source vertices for which outgoing edges are inserted. These are added to an array `to_union`. This is followed by a union operation on these source vertices and their newly inserted adjacent edges. Full path-compression of the union-find tree is applied to finalize the labels which are representatives of the weakly connected components for each vertex.

The incremental WCC kernel implemented in `Meerkat` uses a union-find auxiliary data structure. It largely follows `IterationScheme1` using `UpdateIterator`. Since an `UpdateIterator` iterates also over the partially updated slabs, it is imperative that we ignore those parts of the slab which are populated by the previous incremental updates, for performance. The decremental WCC on GPU is an unsolved problem.

**Table 5** Properties of input graphs. (M=$10^6$, $K = 10^3$)

| Graph | #Nodes | #Edges | Average degree | Max degree | Max diameter | Memory allocation within | |
|---|---|---|---|---|---|---|---|
| | | | | | | SlabHash object | Meerkat object |
| Higgs [14] | 45.6K | 14.9M | 32 | 1259 | 9 | 405 M | 271 M |
| LJournal [15] | 4.85M | 69 M | 14 | 20293 | 16 | 2617 M | 993 M |
| Pokec [16] | 1.63M | 30.6M | 18 | 8763 | 11 | 983 M | 467 M |
| Rand10M | 10 M | 80 M | 8 | 27 | 11 | 5035 M | 1373 M |
| BerkStan [17] | 685K | 7.6M | 11 | 249 | 573 | 491 M | 253 M |
| Wiki-talk [18] | 2.4M | 5 M | 2 | 100022 | 9 | 1345 M | 471 M |
| Wikipedia | 3.4M | 93.4M | 27 | 5333 | 262 | 2063 M | 1017 M |
| Orkut [19] | 3.1M | 234.4M | 76 | 33313 | 9 | 2421 M | 1725 M |
| USAfull [20] | 23.9M | 58.3M | 2 | 9 | 6261 | OOM | 6175 M |

## 6 Experimental Evaluation

We evaluate our implementation for five graph algorithms: Incremental Weakly Connected Components (WCC), dynamic Breadth First Search (BFS), dynamic Single Source Shortest Path (SSSP), dynamic Triangle Counting (TC), and dynamic PageRank (PR). The experimental evaluation was performed on the NVidia RTX 2080 Ti GPU. The GPU is equipped with 11GB of global memory with a memory bandwidth of 616GB/s, and 4352 CUDA Cores (68 SMs and 64 cores/SM). All the implementations were compiled with `-O3` and `-use_fast_math` flags on the `nvcc` version 11.7 compiler.

We compared the performance of static versions of the above algorithms on `Meerkat`, against publicly available dynamic graph data-structures on GPUs: HORNET, GPMA, and FAIMGRAPH. The graph inputs used for the experimental evaluation are presented in Table 5.

### 6.1 Performance of Insert, Query, and Delete Operation

The warp-cooperative work-sharing execution strategy (WCWS) is adopted by `Meerkat` for insert, delete, and query operations. Data is exchanged among threads using warp cooperative functions such as `ballot_sync`, `ffs`, etc. which are fast, as they work only with registers. In WCWS, multiple threads within a warp thread have different tasks (vertices/edges) assigned to them. The warp threads form a queue for processing these tasks (using `ballot_sync`); a task to be processed is collectively elected by the warp (using `ffs`). Each slab occupies `128 bytes`, which closely matches the GPU's $L1$ cache line size. All warp threads perform coalesced vectorized memory accesses on a slab storing a vertex's adjacent neighbours. We chose a load factor of `0.65` for performing the insert, delete, and query benchmarks on `Meerkat`.

The load factor and outdegree of a vertex decides the number of slablists allocated for every vertex, based on its initial degree.

Our incremental batches of edges (in the *insert* operation) are generated randomly and the edges in a batch are not already present in the graph object. The decremental batch of edges is generated by randomly choosing edges of the benchmark graph. For the query benchmark, edges are simply generated in random, without the previously mentioned restrictions.

For an insert $\langle u, v\rangle$ operation, `Meerkat` applies a hashing function on the destination vertex $v$ in order to determine which slab-list of vertex $u$ should be used to store $v$. By distributing the destination vertices among multiple slab lists, hashing implicitly reduces the number of slabs retrieved for checking the existence of a previously inserted edge $\langle u, v\rangle$ to avoid duplicate insertion of edges. New edges are recorded at the end of the chosen slab list, only if the edge was not previously inserted. This requires a traversal till the end of the slab list. If the last slab of the slab list is full, `Meerkat` obtains a new slab from the pool of pre-allocated slabs, by invoking the slab allocator. The new slab is linked to the end of the slab list, and the new edge is recorded in it.

We experimented with different batch sizes for insertion. Figure 4 compares the relative performance of `Meerkat`, against GPMA, HORNET, and FAIMGRAPH, in inserting a batch of $100K$ edges across various benchmark graphs. Across insertion batch sizes of `10K...100K` edges, `Meerkat` on an average performs 8.39×–12.29× better than GPMA, 10.32×–15.66× better than HORNET, and 3.02×–4.51× better than FAIMGRAPH. Figure 5 compares the relative performance of `Meerkat`, against GPMA, HORNET, and FAIMGRAPH, in deleting a batch of $100K$ edges across various benchmark graphs. Across deletion batch sizes of `10K...100K` edges, `Meerkat` on an average performs 2.69×–4.05× better than GPMA, 17.41×–29.37× better than HORNET, and 2.75×–4.34× better than FAIMGRAPH. Figure 6 compares the performance of a query benchmark for a batch of $2^{20}$ edges. On an average, across query batches of $2^{16}$ to $2^{20}$, `Meerkat` performs 5.25×–12.29× better than GPMA, 1.72×–3.93× better than HORNET, and 2.77×–8.81× better than FAIMGRAPH.

In GPMA, each thread in the insertion algorithm is assigned an edge. Each thread identifies an empty location within the leaf segment to insert the edge. The deletion algorithm proceeds to check for the existence of the edge in the leaf segment and invalidates the entry. For insertion, the edge updates are sorted, and the GPU threads are responsible for identifying leaf segments with empty slots. Both insertion and deletion algorithms check for the violation of density thresholds for the leaf segment, and the violations are rectified by rebalancing the segments bottom-up. The rebalancing of the segments is performed at the warp-level, block-level, or device-level depending on the size of the segment until the density thresholds are satisfied at all the levels from the affected leaf segments. This rebalancing of leaf nodes leads to overhead in running time for GPMA.

HORNET migrates the adjacent neighbours of a vertex to a larger edge block if the current block cannot accommodate incoming edges. In the case of deletion, if the number of adjacent edges is smaller than a threshold, the edges are migrated to a smaller block. This migration of blocks adds overhead to the running time in HORNET.

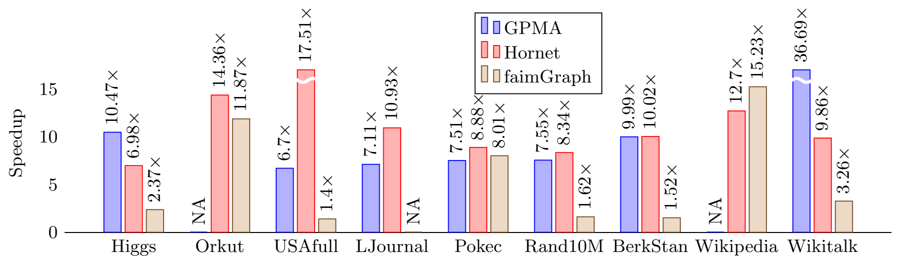


**Fig. 4** Insertion performance—Speedup of `Meerkat` over GPMA, HORNET, FAIMGRAPH for insertion batch size $100K$

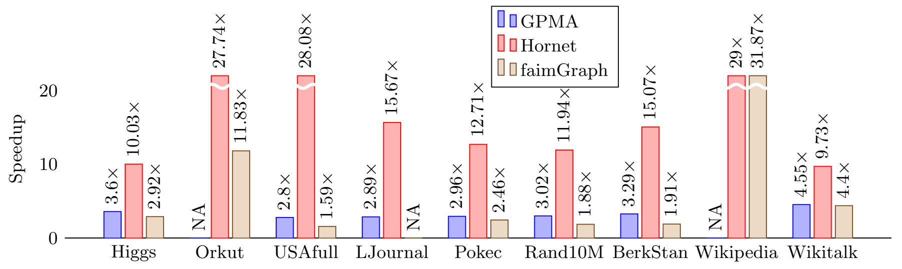


**Fig. 5** Deletion performance—Speedup of `Meerkat` over GPMA, HORNET, FAIMGRAPH for deletion batch size $100K$

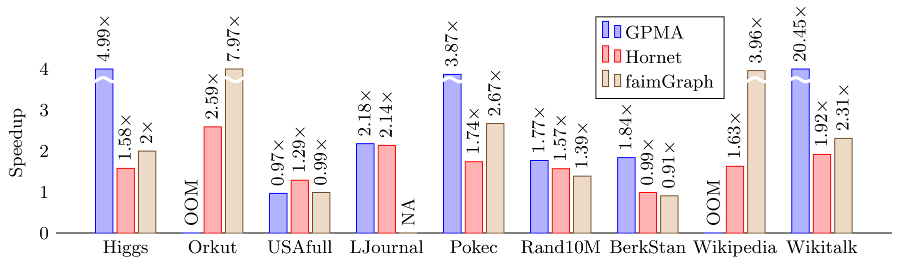


**Fig. 6** Query performance—Speedup of `Meerkat` over GPMA, HORNET, FAIMGRAPH for query batch size $2^{20}$

In FAIMGRAPH, the adjacencies of a vertex are stored in fixed-sized pages. The insertion of an edge is assigned to one worker thread. Each worker thread locks (with a spin-lock) the source vertex before performing the insertion. The adjacencies are inspected for previously inserted duplicate edges, by linear traversal. If no duplication is found, FAIMGRAPH performs an insertion at the first available location in the last page. If existing pages cannot accommodate the edge to be inserted, a new page is allocated and linked to the last page. The spin-lock is released only after a successful insertion, or if a duplicate edge is found. In FAIMGRAPH, once the edges are marked deleted, the last edges from the adjacency list are copied to fill up for the deleted edges. This is followed by sorting of adjacencies and edge compaction.

In `Meerkat`, the hashing function seeks to evenly distribute the adjacencies among the multiple slab lists, and the rebalancing is not necessary. The insertion operation requires adding new slabs once a slab list becomes full. The deletion benchmark shows better performance, as the deletion operation simply flips a valid entry to `TOMBSTONE_KEY`. Unlike the insertion operation, the adjacent neighbour to be deleted could occur anywhere within a slab list. The traversal of the slab list halts once the adjacent edge to be deleted is found.

Compared to HORNET, performance improvement in `Meerkat` is due to better coalesced access, and lack of memory block migration in `Meerkat`. In `Meerkat`, each thread in a warp processes slabs holding neighbours of the same vertex, resulting in better load balance and coalesced memory access. Unlike, FAIMGRAPH, `Meerkat` uses atomics for performing fast insertions in free locations in the slab. The sorting of adjacencies in FAIMGRAPH acts as an overhead compared to `Meerkat`. Sorting of adjacency is not meaningful for a hashing-based graph representation of `Meerkat` with multiple slab lists for a vertex.

## 6.2 BFS and SSSP

The BFS and SSSP computations are programmed in `Meerkat` using two approaches. The VANILLA BFS/SSSP algorithms use 32-bit atomics and their corresponding TREE-BASED implementations use 64-bit atomics.

The VANILLA implementation computes only the shortest distances for reachable vertices from the source vertex, and is thus suitable for static situations. The TREE variant, however, also computes the dependency tree, tracking how these distances have been computed, i.e., the shortest paths. This dependency maintenance is necessary for the correct working of our incremental and decremental SSSP and BFS computations.

The VANILLA BFS and SSSP algorithms on `Meerkat` are implemented in two approaches: a naïve approach named BASELINE, and an improved implementation named CG- SM. The BFS- BASELINE uses level-based traversal of the graph, while SSSP- BASELINE uses an edge-frontier based approach. Both the baseline algorithms update the global memory frontier immediately upon visiting the neighbouring edges of the current frontier. The static/dynamic BFS and SSSP algorithms are programmed in CG- SM.

Populating the frontier for the next iteration $i+1$ requires traversal over the outgoing edges of the destination vertices of the edges in the current iteration $i$. It is prudent to disable hashing for the BFS and SSSP benchmarking since it forces the maintenance of a single slab list for every vertex. The number of slab lists allocated a priori for a vertex is proportional to its degree and varies inversely with the chosen load factor. Increasing the load factor has a direct consequence in improving the slab occupancy, especially in graphs having a high average out-degree (such as Orkut, Higgs, and Wikipedia), and in reducing the total number of allocated slabs to store the initial graph.

Figure 7 shows how the performance of BFS- BASELINE is influenced by various chosen load factors for Orkut (avg. degree 76, 2.38$\times$ slab size), Wikipedia (avg. degree 27, 0.87$\times$ slab size), LJournal (avg. degree 14, 0.44$\times$ slab size), and USA-full (avg. degree 2). We observe 24.76% improvement in the running time across the load factors, for Orkut, Wikipedia, and LJournal. USAfull which has a very low

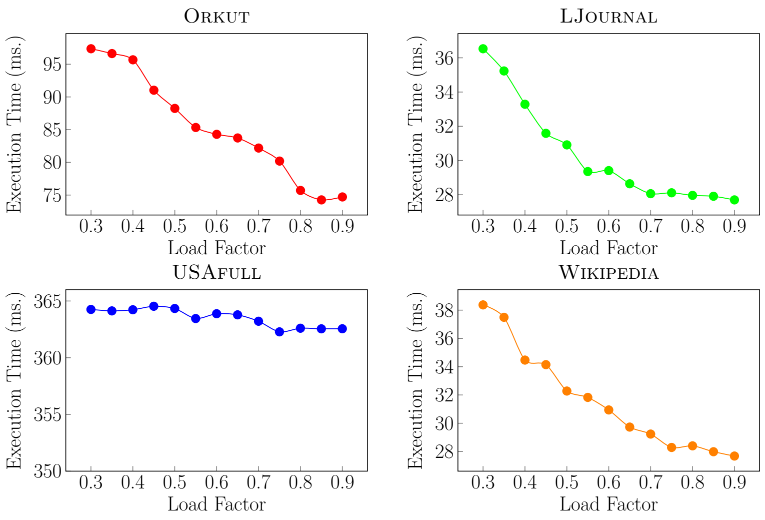


**Fig. 7** Load factor vs Static BFS performance

average out-degree, can accommodate the neighbours of most of its vertices with a single slab (in a single slab list). Hence, it exhibits a meagre 5.68% improvement in performance. On disabling hashing on an unweighted graph representation, the average slab occupancy improves by 24% for Orkut, 14.35% for Higgs, 8% for Pokec, and 5% for LJournal, with 6.26% improvement across all our benchmark graphs, relative to the graph representation with a load factor of 0.7. Disabling hashing for the BFS- BASELINE processing produces an average of 10.78% improvement (up to 28.1%) in performance. Similarly, we observe an average of 9.1% improvement (up to 23.28%) in performance for the TREE- BASED variant. Similarly, disabling hashing for the SSSP- BASELINE benchmark produces an average of 9.9% improvement (upto 35%) in performance. The TREE- BASED variant shows a similar average improvement of 11% (upto 28.95%).

The CG- SM variants improve upon the BASELINE variants by making effective use of cooperative groups and shared memory.

- *Use of* COOPERATIVE GROUPS: We use cooperative groups for implementing our BFS/SSSP kernels. The number of blocks in the grid is equal to the number of streaming multiprocessors (SMs) on the GPU. The number of threads per block is equal to the number of Streaming Processors (SPs) in an SM.
  Every thread block remains resident on the SM throughout the lifetime of the kernel, and a grid-stride loop is used for accessing the elements of the frontier for each iteration. The shared memory for each thread block is equally divided among the warps. Hence, each warp is assigned its own private region of shared memory. The use of cooperative groups provides for grid-wide synchronization within the kernel, thus avoiding the need for explicitly invoking `cudaDeviceSynchronize()` from the host CPU. It is observed that for low-diameter graphs (such as Orkut, LJournal, Pokec, and Wiki-talk), the use of cooperative groups incurs an addi-

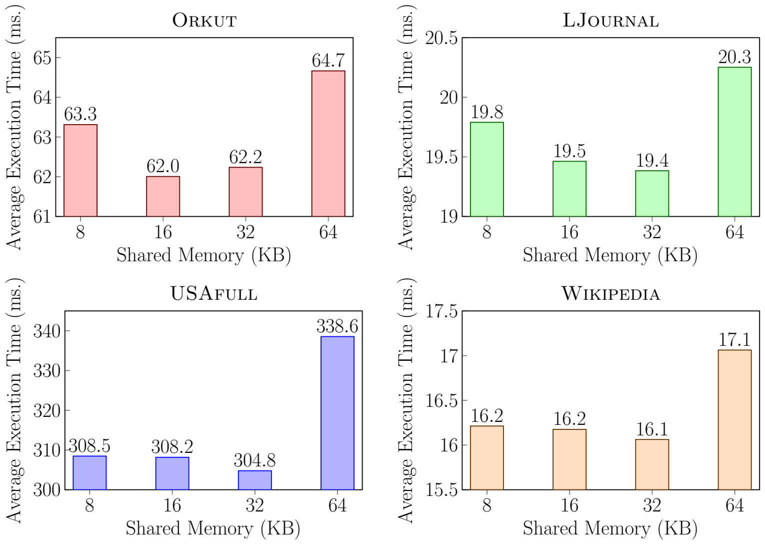


**Fig. 8** Shared memory allocation size vs Static BFS performance

tional overhead of ≈8.67%. However, large diameter graphs have proportionately higher kernel barrier synchronizations and produce an improvement of ≈11.76% on average (9% for USAfull, 8.36% for Wikipedia, and upto 36.36% for Berk-Stan). The use of cooperative groups prohibits a thread block size from exceeding the maximum number of resident threads on an SM, for the correct application of grid-wide synchronization.

- *Privatization of frontier:* It must be recalled that a slab in the unweighted representation can hold up to 31 adjacent vertices, and up to 15 pairs of adjacent vertices and their respective edge weights. Further, the slab occupancy is much lower in low-outdegree graphs such as road networks. Since the out-neighbours and the respective edge weights are enqueued into the SSSP frontier, low slab occupancy leads to poor utilization of global memory bandwidth. In the BASELINE approach, one atomic operation per slab is required to shift the frontier index stored in the global memory. To alleviate this problem, we privatize the frontier into a shared memory partition that is exclusive to a warp. A warp-exclusive partition removes the need for block-level synchronization to overcome hazards. Each warp enqueues the frontier edges into its shared memory partition, and flushes them into the global memory frontier on exhaustion. Every flush operation requires only a single atomic operation to advance the frontier limit. The writes from the shared memory partition to the global memory frontier are performed as a sequence of coalesced writes with the help of a warp-stride loop utilizing full memory bandwidth. Further, a private partition ensures that warps can independently be either in the frontier traversal or private shared-memory partition flushing mode.

Our CUDA compute 7.5 capable GPU allows for carving out a maximum of 64KB shared memory out of the unified shared memory-L1 cache for each thread block. However, as shown in Fig. 8, we have discovered that an allocation of 32KB of shared

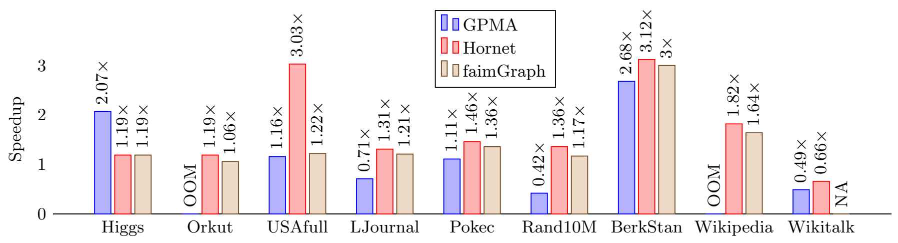


**Fig. 9** Static BFS—Speedup of Meerkat over GPMA, HORNET, FAIMGRAPH

memory delivers optimal performance for most of our benchmark graphs. Since the operation of flushing a warp-private frontier cache into the global memory is a memory-intensive operation, a warp performing a global memory write is scheduled out until the write completes. A large private shared memory impacts the performance negatively, if several warps become memory-bound in flushing their private shared memory frontiers into the global memory, at the same time. A similar situation also bodes for small warp-private frontier cache sizes. On an average, BFS- CG- SM is 1.39× (upto 1.83×) faster than BFS- BASELINE. Similarly, SSSP- CG- SM performs 1.17× (upto 1.33×) better than SSSP- BASELINE, with 32kB shared memory allocated per thread block.

### 6.2.1 Static BFS and SSSP

Figure 9 compares the performance of publicly available implementations of the static BFS algorithm in HORNET, GPMA, and FAIMGRAPH, with Meerkat's BFS- CG- SM variant. HORNET's BFS algorithm implementation uses an iterative level-based approach using a pair of vertex frontiers: one for the traversal of the current level, and the other for holding the unvisited vertices of the subsequent level. A similar approach is also adopted by the FAIMGRAPH's naïve implementation. The FAIMGRAPH framework also provides three other variants (namely BFS- DYNAMIC-PARALLELISM, BFS- CLASSIFICATION, and BFS- PREPROCESSING) that improve over the naïve implementation. These implementations make use of CUDA's dynamic parallelism for enabling level-based traversal of graph edges from the source vertex. BFS- CLASSIFICATION maintains separate degree-specific frontier queues for small-degree, medium-degree, and large-degree vertices; the enqueuing of unvisited vertices is performed into only a single RAW queue. The vertices from this RAW frontier queue are classified into the degree-specific frontier queues. The traversal in the subsequent iteration is performed individually on these degree-specific queues. Since every thread is assigned a unique vertex for the traversal of its neighbours, the classification of vertices into degree-specific frontier queues seeks to reduce warp divergence during frontier enqueue operation and efficient dynamic parallelism within the kernel. The BFS- PREPROCESSING is similar, except that it eliminates this RAW frontier queue, and maintains separate degree-specific pairs of frontiers for enqueueing and traversal. MEERKAT's BFS- CG- SM implementation on an average performs 1.48×, 1.24×, and 1.68× better than FAIMGRAPH, GPMA, and HORNET respectively. GPMA goes out-of-memory (OOM) for two large graphs, namely Wikipedia and Orkut, on our 11GB GPU. GPMA stores every edge as a 64- BIT key-value pair within one of the

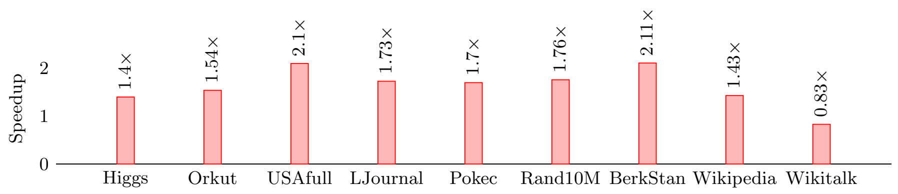


**Fig. 10** Static SSSP—Speedup of `Meerkat` over HORNET

memory segments in the PMA array. Furthermore, GPMA over-subscribes memory allocation for its PMA arrays in the GPU's global memory (beyond the CSR graph storage requirements), for maintaining the occupancy threshold invariants for each of its memory segments.

Static SSSP of HORNET, and `Meerkat`'s SSSP- CG- SM are compared in Fig. 10. Both versions follow an iterative processing using a pair of frontiers. The public source code for GPMA and FAIMGRAPH are not available, and are hence, missing from this comparision.

`Meerkat`'s SSSP- BASELINE is on average, 1.32× (up to 1.85×) faster than HORNET's implementation. Likewise, MEERKAT's SSSP- CG- SM outperforms HORNET's SSSP implementation by 1.62× (upto 2.11×).

The privatization of the frontier queue in `Meerkat` always ensures coalesced writes into the global memory queue. HORNET, GPMA, and FAIMGRAPH, in contrast, write directly to their global memory frontier. To obtain the frontier queue's offset, HORNET and FAIMGRAPH perform an exclusive scan on the number of unvisited adjacent vertices for each thread in the block; each thread within the block obtains a unique local offset within the global frontier to write its share of frontier vertices. Only one atomic operation per thread block is necessary to shift the frontier's end-pointer. These operations require block-wide synchronization with the help of `__syncthreads()`. Further, the offset obtained by each thread does not permit warps within the thread block to perform coalesced writes into the global memory frontiers. In contrast, in `Meerkat`, a shared memory partition is exclusive only to a warp of threads. This avoids the need for block-wide synchronization while a warp flushes its exclusive shared memory partition into the global memory frontier.

For traversal of neighbours, HORNET divides the edges to process equally among thread blocks, and further, equally among the threads within a thread block. Warp divergence is avoided significantly, and load balance within the warp is achieved as the threads have approximately an equal number of edges to process. Each thread, in fact, stores offsets to the edges to be processed, and not the edges themselves. Since the edges are divided into sequential chunks among threads, it results in divergent memory access among warp threads. Further, extra memory access is required to translate offsets to actual edge data. For FAIMGRAPH's BFS traversal of neighbours, each thread is assigned a unique vertex from the frontier queue. Though traversal of neighbours is achieved with FAIMGRAPH's iterator abstractions, their flexibility to iterate over neighbours of different vertices within a warp leads to uncoalesced reads from the frontier queue. Although the average slab occupancy in `Meerkat` is about 36.61% across all our benchmark graphs, our iterators always perform coalesced memory accesses to retrieve adjacent vertices using the warp-cooperative work strategy. `Meerkat`'s iterator abstraction enforces that a warp of threads always refers to a unique slab, for a specific vertex.

### 6.2.2 Dynamic BFS and SSSP

Incremental/Decremental BFS and SSSP algorithm implementations are missing from the publicly available source-codes for HORNET, GPMA, and FAIMGRAPH. Moreover, we have demonstrated that our `Meerkat`'s static BFS and SSSP algorithms perform better than those publicly available with HORNET, GPMA, and FAIMGRAPH. Hence, we compare the performance of our incremental/decremental BFS and SSSP algorithms against their static counterparts in `Meerkat` itself.

The TREE- BASED BFS and SSSP computations in `Meerkat`, in contrast to the VANILLA implementations initialize tree-nodes (a 64-bit sized $\langle \texttt{distance}_{\texttt{SRC}}, \texttt{parent}\rangle$ ordered pair with distance stored in most significant 32 bits) and updates the pair using 64-bit atomics. This is necessary for setting up the initial data structures for incremental and decremental variants of our BFS and SSSP implementations on `Meerkat`. The TREE- BASED BFS- CG- SM has an average overhead of 35.27% over its VANILLA counter-part, as seen in the execution time. A similar overhead of ≈18% was seen with the case of TREE- BASED SSSP- CG- SM. Figure 11a shows the speedup of incremental/decremental BFS over static BFS in `Meerkat`. The speedup for incremental/decremental SSSP over static SSSP in `Meerkat`, is shown in Fig. 11b. The batch size ranges from $10K$ to $100K$. The incremental BFS and SSSP are bound to be faster than the decremental algorithm. The incremental BFS and SSSP are performed by choosing the input batch of edges as the initial frontier and iterative application of the static algorithm to recompute the TREE. The decremental variant involves invalidation of the affected vertices in the TREE, the propagation of invalidation up the TREE, computation of the initial frontier from unaffected vertices and re-computation of the TREE invariant by iterative application of the static algorithm. While the *speedup decreases empirically* for the increase in the batch sizes, *anamolies* are exhibited for certain batch sizes (for example, in BerkStan, batches of $40K$ edges show lower average speedup than the batches of $60K$ edges). The speedups show a dependence on the batch size and the edges of the batch themselves. It must be recalled that, for our experimental evaluation, the batch edges are randomly chosen from a uniform distribution. Certain incremental batches induce the reachability of more vertices which were previously unreachable. This increases the size of our spanning TREE comprising of vertices reachable from the source, and consequently, the BFS/SSSP distance. This is particularly observed in BerkStan for incremental SSSP (as shown in Fig. 11b). With exception to USAfull and BerkStan, the execution times of repeated application of the static algorithm were on an average 7.3% and 5.6% lower than static running time on the original graph, for incremental BFS and incremental SSSP respectively. For USAfull and BerkStan, this difference was close to 80% and 71% respectively. In the case of incremental BFS on USAfull, we observed that there was 11.53× decrease in the average distance of vertices reachable from the source vertex, after the addition of first `10K`, and nearly 2× decrease from the first to the tenth batch. USAfull has a single large connected component, and hence no significant increase in the number of

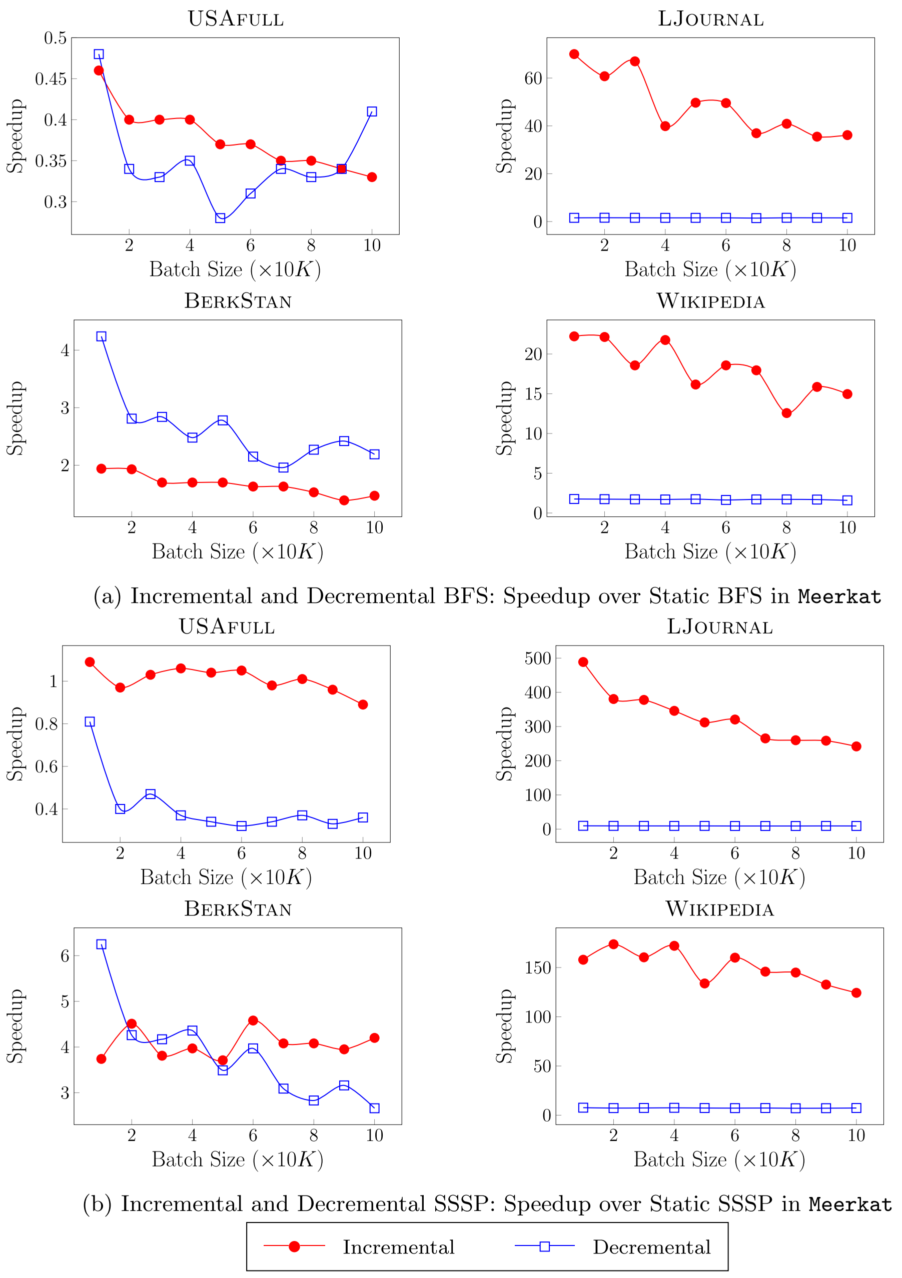


**Fig. 11** Incremental/Decremental BFS and SSSP

reachable vertices was observed. In the case of BerkStan, while the average distance of reachable vertices decreased from 11.7 to 8.58 for our sequence of ten incremental batches of 10K, we observed that the number of reachable vertices increased from $\approx 460K$ to $\approx 591K$. Due to this graph topology, the speedup for incremental BFS/SSSP was much lower for the USAfull and the BerkStan graphs compared to other graphs. We have not seen any significant increase in the number of reachable vertices or a decrease in the average distance of reachable vertices for other benchmark graphs.

Like the case of incremental BFS and SSSP, the decremental counterparts also show dependence on the nature of batch edges, while showing lower speedups *empirically* with increasing batch sizes. A batch of a certain size is likely to show a lower speedup if it contains a TREE edge, with one endpoint close to the source, and the other end-point being the root of a large sub-TREE (several vertices can trace their distance computation through this intermediate vertex). If the vertices in this invalidated sub-TREE are still reachable, re-computation of their invalidated distances from the source vertex leads to a lower speedup. In the case of decremental BFS and SSSP, the number of edges in the dependence tree that have been invalidated depends on the average in-degree of the vertex. We observed that for low average in-degree graphs, the likelihood of tree edges being invalidated was higher than those of high in-degree graphs. For example, in the case of decremental BFS for a sequence of ten 10K batches, for USAfull (with an average in-degree 2), an average of 38.97% of the decremental batch were tree edges, while it was 0.769% of the decremental batch for Orkut (with an average in-degree 76). It must be understood that the depth of the dependence tree is the BFS distance. Smaller tree depth (BFS distance) and large average degree favour only fewer vertices to be invalidated. In our observation of decremental BFS for 10K batches, we saw an average of 0.23K vertices for Orkut, 0.4K for Wikipedia, 1K for LJournal, 3.94K for BerkStan, 6K vertices for Rand10M, whose distances were invalidated in the TREE, while it was an average of 9.54M vertices for USAfull, after each batch. This explains why USAfull performs poorly with our decremental BFS/SSSP algorithm. In the case of BerkStan, we have observed a decrease in the average distance of vertices reachable from the source vertex, for successive decremental batches, while other graphs have shown a marginally increasing trend in the average distance for successive decremental batches. This is because BerkStan has several critical edges: the presence of critical edges in the decremental batches produces new components whose vertices are unreachable from the source vertex. In other graphs, the vertices continued to remain reachable from the source, but with alternativew longer shortest paths. The number of reachable vertices for BerkStan reduced by $\approx 2\%$ after ten batches; Rand10M and Orkut did not show any decrease as the single large graph component continued to remain connected; the decrease was on an average 0.047% (upto 0.18% for USAfull) for other graphs.

### 6.3 PageRank

For our experimental evaluation of PageRank, we have set the damping factor to be 0.85, the error computation with L1- NORM as the convergence strategy, and the error margin to be 0.00001. The computation of PageRank (See Algorithm 5) involves the traversal of neighbours of each vertex, along their incoming edges. Hashing was disabled in the PAGERANK implementation as done in SSSP and BFS to improve performance.

Disabling hashing improves the slab occupancy, especially in graphs with a higher average in-degrees. For low average in-degree graphs such as USAfull, Rand10M, and Wiki-talk, there is no performance improvement observed, as disabling hashing has virtually no effect: most vertices owing to their low in-degree have single slab lists. However, for large average in-degree graphs such as Orkut, and Wikipedia, disabling hashing produces a speedup of about $1.36-1.62\times$ in the static PageRank running time.

### 6.3.1 Static Pagerank

Figure 12 compares the performance of static PageRank on `Meerkat`, with that of HORNET and GPMA. It is observed that in seven out of nine graphs(that is, except Rand10M, Higgs), `Meerkat` performs $1.18-2.93\times$ (with an average of $1.89\times$) faster than HORNET. The PageRank implementation on both `Meerkat`, GPMA, and HORNET are traversal-based algorithms. Each iteration applies the computation of PageRank on all vertices. We were unable to compare GPMA's PageRank implementation for ORKUT and WIKIPEDIA due to out-of-memory (OOM) errors. GPMA pre-allocates larger arrays to leave empty slots to accomodate new neighbours for a vertex, and to satisfy its internal thresholds for maintaining tree-balancing. GPMA's array allocation for Orkut and Wikipedia exceeded the available global memory on our GPU. `Meerkat` performed on an average, $8.16\times$ (upto $22.73\times$ for WIKI- TALK) better than GPMA's implementation for other graphs.

For our comparison, convergence in all the benchmark implementations is achieved when the L1- NORM between PAGERANK vectors $\mathbf{PR}_i$ and $\mathbf{PR}_{i-1}$ for iterations $i$ and $i-1$ respectively is less than the error margin. The performance improvement in `Meerkat` can be attributed to our efficient iterators performing coalesced accesses in retrieving adjacent vertices. While HORNET attempts to avoid warp-divergence, its traversal mechanism does not perform coalesced accesses.

GPMA attempts to perform a clever load-balancing for the traversal of neighbours based on the average degree of vertices. The threads within a block are grouped by the smallest multiple of two greater than the average degree, or the warp size, whichever is smaller. Each such group of threads $t$ is assigned a particular vertex $v$ in a grid-stride

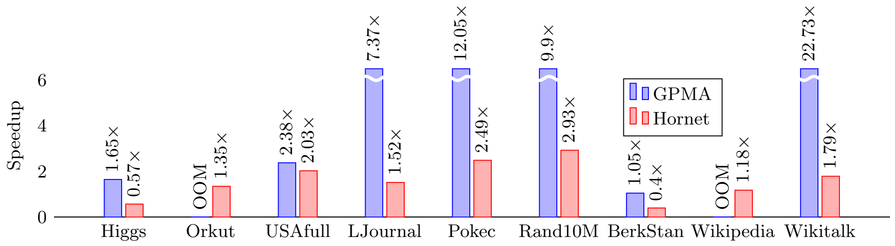


**Fig. 12** Static PageRank—Speedup of `Meerkat` over GPMA, HORNET

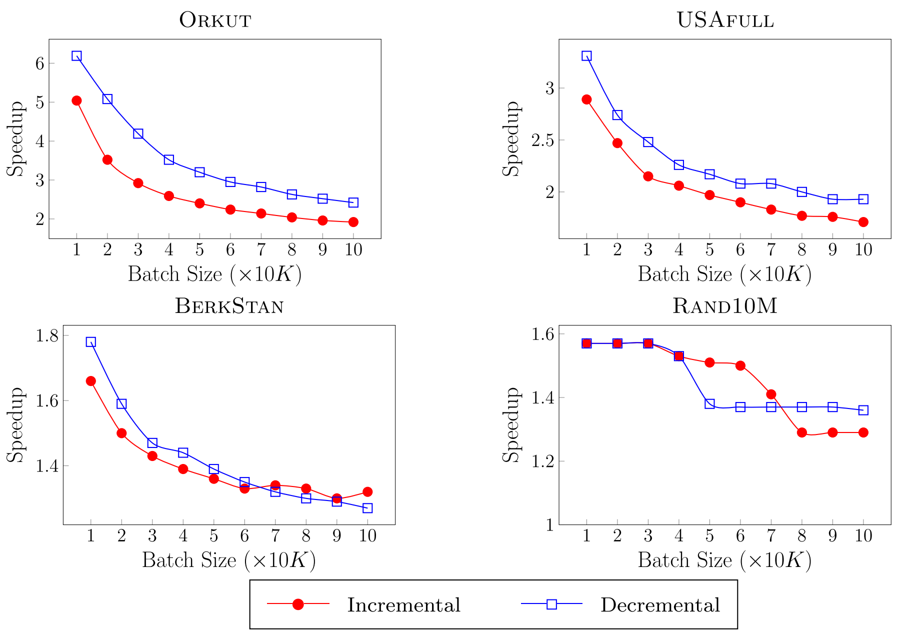


**Fig. 13** PageRank Incremental/Decremental—Speedup over Static PageRank in `Meerkat`

loop. Such a thread-group $t$ is responsible for the traversal of neighbours of $v$: each thread in the thread-group $t$ accesses a unique neighbour of $v$. Full coalesced access may be possible if the size of $t$ is equal to the warp size. The memory accesses become increasingly diverged within a warp if the average degree of vertices becomes smaller. In the case of Higgs, the ratio of the number of edges to the number of vertices is equal to the warp size, and a full warp is used for the traversal of the adjacencies of every vertex, and approximates the warp-cooperative work execution strategy in `Meerkat`. Hence, `Meerkat` has a small speedup over GPMA for this graph. For WIKIPEDIA, only four threads are assigned for every vertex. This graph has few vertices with a large out-degree, even though its average degree is small. Hence, GPMA incurs a huge execution time. In graphs such as LJournal, Pokec, and Rand10M, a warp becomes memory-bound, before it performs a PAGERANK for a relatively short duration. Firstly, each thread group $t$ within a warp issues its memory requests at the same time to fetch the neighbours of their respective vertices. Secondly, it must be remembered that the ratio of the PageRank of an incoming neighbour to its out-degree is invariant for an iteration. While `Meerkat` and HORNET pre-compute and cache this ratio for every vertex for every iteration, GPMA's implementation does not. Since every thread within a warp could potentially have a different in-neighbour, every thread performs two uncoalesced memory accesses for the computation of this ratio.

### 6.3.2 Dynamic PageRank

The incremental and decremental algorithms are identical: the same static PageRank algorithm is applied on the entire graph after performing insertion/deletion of edges, respectively. The Fig. 13 speedup for incremental/decremental PageRank over static PageRank computation for batch sizes, ranging from `10K` to `100K`. We make two important observations in our evaluation. Firstly, the number of vertices, whose PageRank values are invalidated, increases with the batch size.

Hence, we observe an decreasing trend in the speedup as the batches increase in size. We have also observed that decreasing trend in the speedup is due to the increase in number of iterations to reach convergence, with growing batch sizes. In Fig. 13, Orkut, USAfull, and BerkStan have consistently shown a decreasing trend in the speedup for incremental and decremental batches, as number of iterations to achieve convergence increases with increasing batch sizes. For Rand10M, we observe that some flattening of the speedup curve. This is because of the number of iterations to achieve convergence remaining stable with increasing batch sizes. For an example sequence of *ten* `10K` incremental (decremental) batches of random edges, we observed that Orkut achieved convergence with $\approx 20\%$ ($\approx 13\%$) of iterations required for the static variant. Whereas, the Rand10M converged in $\approx 64\%$ of iterations required for the static variant, for both incremental and decremental algorithms, registering the slowest speedup. It must be recalled that Rand10M has a low average out-degree compared to Orkut. According to equation 1, for a batch edge $u \rightarrow v$, the change in vertex $u$'s PageRank contribution to vertex $v$ is higher with Rand10M than Orkut, leading to more iterations to achieve convergence, and hence a lower speedup.

Secondly, since all vertices of a graph participate in the computation of incremental and decremental PageRank, the per iteration running time also depend on the number of vertices. Hence, the average per iteration running time per iteration is higher for graphs with a large number of vertices, such as USAfull and Rand10M.

Since, a warp performs the PageRank computation of one vertex at a time, each warp fetches an average of two neighbours for the PageRank computation, in an iteration. USAfull has the highest number of vertices and a low average degree. This combined effect make USAfull show the highest running time per iteration among all graphs. A similar effect is also shown with Rand10M. Owing to their small diameters, Orkut, LJournal, and Pokec converge with fewer iterations compared to static PageRank.

## 6.4 Triangle Counting

Hornet and faimGraph provides implementation of static triangle counting. The implementation of triangle counting in Hornet and faimGraph pre-processes the input batch, sorting the edges so that adjacent neighbours of every vertex can be accessed in ascending order before running the triangle counting algorithm.

This ordering of edges is beneficial for performing intersections of the adjacencies of the endpoints of an edge. In a dynamic setting, such an algorithm will require sorting of adjacencies, and re-construction of the graph object, before each triangle-count recomputation. *Hence, algorithm with sorted adjacencies will not scale in dynamic triangle counting.*

### 6.4.1 Static Triangle Counting

We compare two different implementations of the static triangle counting algorithm on `Meerkat`:

TC- Query: This naive approach iterates over every edge $(u, v)$: for every edge $(u, w)$, it checks the existence of the edge $(v, w)$ using the SearchEdge() for computing the intersection of adjacencies of vertices $u$ and $v$.

Enabling hashing distributes the slabs among multiple slab lists; only the slab list that could potentially accommodate the search vertex can be inspected. This reduces the number of slabs to inspect while performing SearchEdge() operation during the intersection operation.

TC- Sorted: In this method, the adjacencies of the input graph are first sorted. Hashing is disabled and only one slab list is allocated for every vertex to store its adjacencies in sorted order. Each warp thread is assigned one edge at a time. Consequently, each thread holds a pair of iterators for each end-point of the assigned edge, for the traversal of their respective adjacencies, during the intersection operation.

Initially, the shared memory allocation for each warp thread is initialized with the head (first) slabs of the slab lists for both end-points of the edge, in a warp-cooperative fashion. The intersection of the adjacencies by a warp thread is performed by a linear scan of both slabs fetched into the shared memory. If one of the slabs for all threads has been exhausted by linear scan, the successor slabs are fetched using warp-cooperative work execution.

For five out of nine graphs (Orkut, LJournal, Pokec, Rand10M, Wiki-talk), we see an average speedup of 2.78× (upto 6.58×) for TC- Sorted over TC- Query. In TC- Query, the SearchEdge() executes in a warp-cooperative fashion: a work queue maintains a list of outstanding threads whose edges are left to be queried.

The presence of hashing mitigates the number of slabs inspected, but cannot enforce sorted property among adjacent vertices. However, in TC- Query, all the warp threads perform independent intersection operations on their respective pairs of slab lists, yielding higher throughput. As the USAfull has a very low average degree, both the SearchEdge() in TC- Query and the linear scan in TC- Sorted will fetch a similar number of slabs for the intersection operation. USAfull has similar performance with both TC- Query and TC- Sorted. TC- Query performs 5.33× and 3.57× better than TC- Sorted, for BerkStan and Wikipedia, respectively. This is a result of a few vertices having a large degree in these graphs, leading to a few warp threads performing long-tail intersection operations.

Figure 14 compares the performance of the static triangle counting algorithm on `Meerkat` against Hornet and faimGraph. It is observed in our experimental evaluation that Hornet performs on an average 23.2× (upto 53.06×) faster than that of `Meerkat` on our benchmark graphs. Similarly, faimGraph performs on an average 3.38× faster than `Meerkat`.

In `Meerkat`, the neighbours of a vertex are stored in fixed-size slabs. While these neighbours are contiguous within a slab, the slabs themselves are not contiguous with

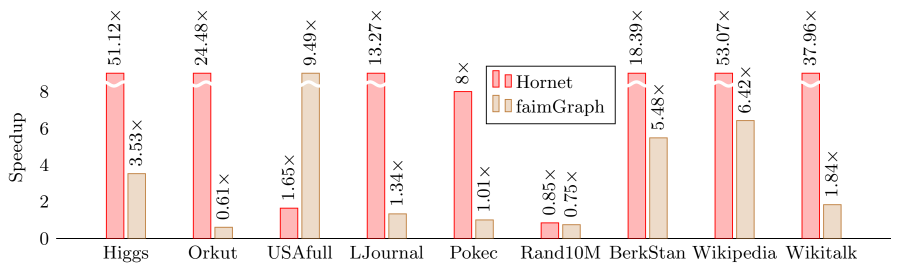


**Fig. 14** Static Triangle Counting—Slowdown over HORNET, FAIMGRAPH

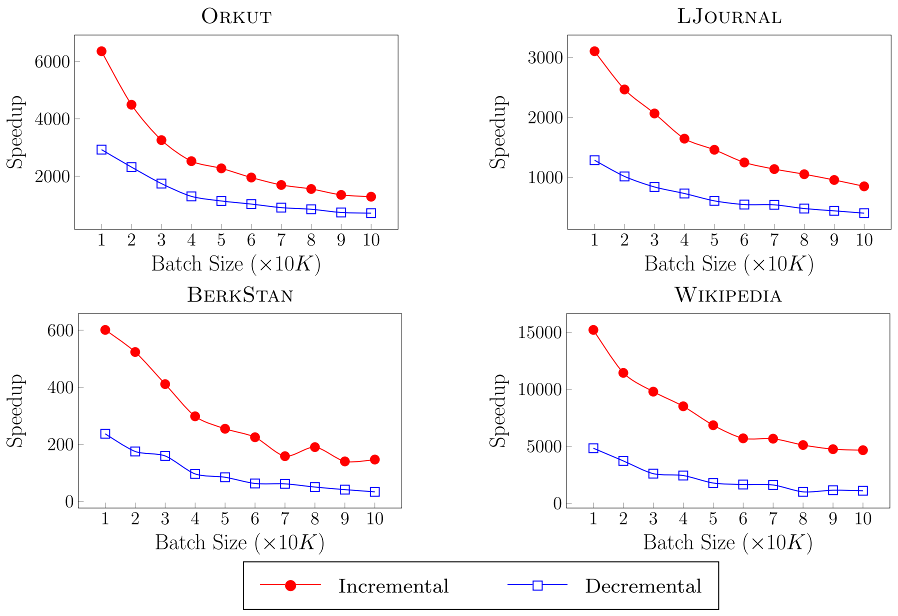


**Fig. 15** Triangle Counting incremental/decremental—Speedup over static triangle counting in `Meerkat`

each other. In HORNET, all the neighbours of a vertex are contiguously available within an edge block, whose size is the smallest power of two greater than the number of adjacent neighbours.

`Meerkat` cannot use efficient methods which HORNET can use due to this lack of ordering of edges. *The sorting adjacencies is useful in a static setup, but will lead to high overhead in the dynamic triangle couting algorithm to maintain the sorted order of edge adjacencies.* HORNET does not have an implementation of dynamic TC.

### 6.4.2 Dynamic Triangle Counting

Fig. 15 shows the speedup of our incremental/decremental algorithms over the static algorithms on `Meerkat`. Across the benchmarks, superlative speedups are observed since, for each batch, the static algorithm counts the number of triangles by performing

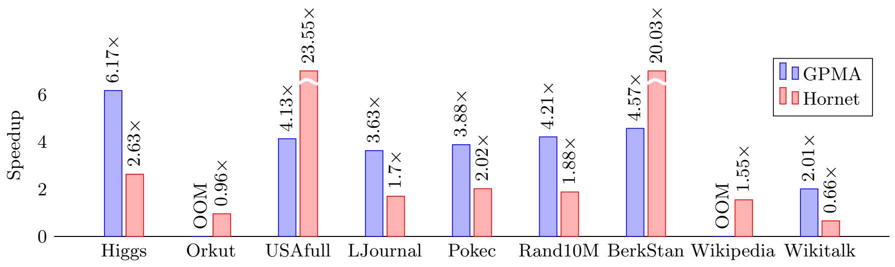


**Fig. 16** Static WCC—Speedup of Meerkat over GPMA, HORNET

an intersection for the adjacencies of both end-points for every graph edge, while the dynamic algorithm performs intersection only for the end-points of the edges in the batch. The speedup observed is very large if the batch size is very small compared to the number of edges in the graph. Hence, large graphs such as Orkut, LJournal, Rand10M, and Wikipedia enjoy very high speedups compared to the repeated application of the static algorithm.

## 6.5 Weakly Connected Component (WCC)

We evaluate the performance of the static WCC algorithm on `Meerkat` against HORNET and GPMA, followed by the performance of incremental WCC in `Meerkat` for various optimizations. FAIMGRAPH's public repository does not provide a complete implementation for weakly connected components, and is thus, excluded from the comparison.

STATIC WCC IN `Meerkat`: The Fig. 16 compares the performance of static WCC on `Meerkat` against HORNET and GPMA. HORNET uses a modified-BFS like algorithm discovering connected components, using a two-level queue. With a two-level queue, the `insert()` and `dequeue()` operations are performed on two separate queues. In the first step, the discovery of the largest connected component is attempted using a BFS from the source vertex with the help of a two-level queue. All the reachable vertices are marked with the same color. Then all unvisited vertices are incrementally assigned a unique color.

This iterative process continues until all the endpoints of the edges have the same color. In `Meerkat`, the static WCC implementation uses the union-find approach for discovering weakly-connected components. It performs a single traversal through all the adjacencies of the graph: it uses the UNION- ASYNC strategy [21] for the UNION() operation for the adjacent edges discovered, and full path compression for determining the representative elements for the vertices in FIND() operation. The GPMA's implementation also uses the union-find approach: a grid-stride loop scans over the array of edges: for edge $(u, v)$, it *hooks* the parent (the representative element) of one of the end-points (say $parent(u)$) as a child of the parent of the other end-point (say $parent(v)$), if both the end-points do not have identical representative parent vertices.

We observe that while `Meerkat` performs 6.11× on an average across all our input graphs, the speedup against HORNET is lower if there are vertices with very large

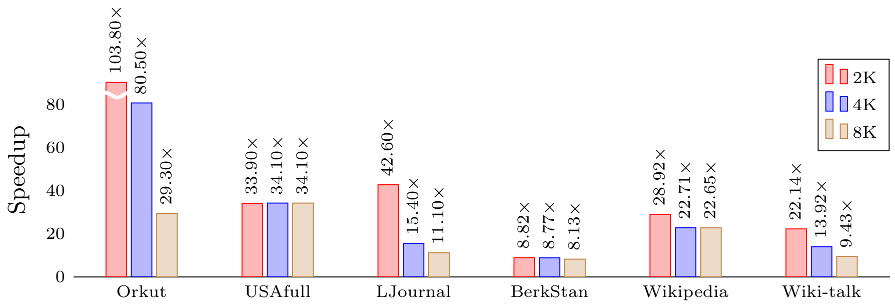


**Fig. 17** Incr. WCC using UPDATEITERATOR + SINGLE SLAB LIST—Speedup over NAIVE

out-degree. This is observed in graphs such as Orkut, LJournal, and Wikipedia. This is because a large-out degree vertex will cause many vertices to be enqueued into the BFS frontier queue, thereby improving parallelism. However, for networks such as USAfull, BerkStan, with high diameter, the BFS-approach in HORNET performs significantly worse compared to `Meerkat`. `Meerkat` performs 4.09× on an average across all our input graphs over GPMA(except for Orkut and Wikipedia which could not fit on our GPU global memory on GPMA).

The parent hooking operation used by GPMA is a simple non-atomic update. The union-find path-compression kernel is executed in parallel to the parent-hooking kernel. In this implementation a FIND($v$) operation on a vertex $v$ may not return the representative element of its weakly connected component. Thus several iterations of parent-hooking followed by path compression may be required until a fixed point of FIND($v$) is reached for every vertex $v$. Each iteration inevitably involves a linear scan of all edges in the GPMA graph. This results in high execution time.

INCREMENTAL WCC IN `Meerkat`: Incremental WCC of `Meerkat` is evaluated with two schemes. (i) *Naive*: traverses through all the slab lists as it is ignorant about the location of the new updates. Hence, it is expected to be the least performant implementation. This algorithm is identical to the naive static WCC algorithm on SlabGraph. (ii) *UpdateIterator* + *Single slab list*: Evaluates the *UpdateIterator* approach in the absence of hashing. Intuitively, this should ensure that a warp operating on an *UpdateIterator* would see more updates in single memory access while traversing a slab list. This method produced good speedup with respect to the *Naive* variant..

Figure 17 compares the performance of *UpdateIterator* + *Single slab list* against the *naive* scheme. The running time of *naive* scheme is proportional to the number of edges present in the graph representation. The *UpdateIterator's* optimized processing iterates over only the updated slabs (See Sect. 4.3.1). Therefore, the running time is proportional to the number of slabs holding new updates. The speedup over the NAIVE scheme is determined by the ratio of total slabs in the graph to the number of slabs holding the newly inserted edges. The *UpdateIterator* process only slabs with new edges while the NAIVE scheme process all the slabs.

The use of *UpdateIterator* with the vertex flag, and the allowance of multiple slab lists seems to have lower performance than the use of a single slab list for high-degree graphs that follow the *power-law* distribution: Orkut, Wikipedia, and Wiki-talk. In

**Table 6** Related works summary

| Related work | (1) | (2) | (3) | (4) | (5) | (6) |
|---|---|---|---|---|---|---|
| (D) STINGER [7] | ✓ | ✗ | Adjacency list of edge-blocks | ✗ | ✗ | ✗ |
| (F) KickStarter [11] | ✓ | ✗ | (n/a) | ✗ | ✗ | ✗ |
| (F) LiveGraph [22] | ✓ | ✗ | Edge-block arrays + persistent storage | ✓ | ✓ | ✗ |
| (D) GraphTinker [23] | ✓ | ✗ | Adjacency list of edge-blocks | ✗ | ✗ | ✗ |
| (F) GraphOne [24] | ✓ | ✗ | Adjacency List + persistent storage | ✓ | ✓ | ✗ |
| (D) Teseo [25] | ✓ | ✗ | Packed-memory Array (PMA) + B+-Tree | ✓ | ✓ | ✗ |
| (D) Sortledon [26] | ✓ | ✗ | Adjacency skiplist of edge-blocks | ✓ | ✓ | ✗ |
| (F) GraphFly [27] | ✓ | ✗ | Dependence-flow tree | ✗ | ✗ | ✗ |
| (F) Terrace [28] | ✓ | ✗ | Edge-block arrays + PMA + B+-Tree | ✗ | ✗ | ✗ |
| (F) Aspen [29] | ✓ | ✗ | Compressed functional tree | ✗ | ✗ | ✗ |
| (F) CommonGraph [30] | ✓ | ✗ | Compressed Sparse Row | ✗ | ✓ | ✗ |
| (D) cuSTINGER [5] | ✗ | ✓ | Adjacency list of edge-blocks | ✗ | ✗ | ✗ |
| (D) Hornet [4] | ✗ | ✓ | Adjacency list of edge-blocks | ✗ | ✗ | ✗ |
| (D) GPMA/GPMA+ [8] | ✗ | ✓ | Packed-memory array | ✗ | ✗ | ✗ |
| (D) LPMA [31] | ✗ | ✓ | Packed-memory array | ✗ | ✗ | ✗ |
| (D) aimGraph [32] | ✗ | ✓ | Edge-block arrays | ✗ | ✗ | ✗ |
| (D) faimGraph [3] | ✗ | ✓ | Adjacency list of edge-blocks | ✗ | ✗ | ✗ |
| (D) SlabGraph [2, 6] | ✗ | ✓ | Hash Tables | ✗ | ✗ | ✓ |
| (F) EGraph [33] | ✗ | ✓ | see [34] | ✗ | ✓ | ✗ |

(D): Data Structure (F): Framework (1): CPU (2): GPU (3): Data Structure Used for Graph Representation (4): Transactional Support for Updates (5): Snapshot-based Graph updates (6): Warp Cooperative Work Sharing

the presence of multiple slab lists, the *UpdateIterator* must sequentially probe if the *update* flag for a slab list is set for every slab list of the source vertex. This overhead is overcome in the use of a single slab list by subsuming the function of the *update* flag of a slab list within that for the source vertex. In high-degree graphs such as social networks (namely, Orkut, Wikipedia, and Wiki-talk), the *UpdateIterator* over a single slab list overcomes previously inserted vertices, with the updates restricted contiguously to a single slab list, resulting in marginal increase in performance over multiple slab lists.

## 7 Related Work

In the recent past, multiple works have addressed the challenges dealing with dynamic graphs on CPUs, GPUs.

*Dynamic Graph Data Structures on CPU*: The Packed Memory Array (PMA) [35] is a sequential data structure for dynamic graphs. PMA is maintained as a self-balancing binary tree [36] in which the memory is divided into multiple leaf segments, and the non-leaf segments identify the memory occupied by their children segments. Graph-

Tinker [23] is a CPU-based dynamic graph data structure that overcomes STINGER's [7] edge query performance (required for insertion/deletion operations) using Robin-Hood and tree-based hashing. GraphTinker uses one of the store-and-static computation (full-graph processing) and the incremental-computation mode for every iteration in processing graph algorithms. If the ratio of active vertices to total edges processed exceeds a threshold, full graph processing is chosen, otherwise incremental computation is performed. Teseo [25] extends the PMA-based graph data structure with transactional semantics for graph updates. Sortledon [26] proposes a sorted adjacency-list-based transactional graph data structure providing for concurrency control for graph update, versioned storage and consistency guarantees.

*Dynamic Graph Frameworks on CPU*: KickStarter [11] formalizes a transitive dependence-tracking approach for computing monotonic graph algorithms (such as CC, and SSSP) for streaming graph applications. *A similar approach is used in dependence tree-based dynamic BFS and SSSP algorithms in `Meerkat`.* CommonGraph [30] extends the KickStarter framework: to avoid expensive deletion and mutation operations in the graph, it considers a subset of edges common to all graph snapshot versions (known as CommonGraphs) and translates edge deletion operations to insertions. LiveGraph [22] proposes a new data structure called transactional edge log based on an optimized OLTP protocol, for concurrent edge queries and edge insertions. GraphOne [24] attempts to isolate the data store from stream/batch analytics and supports analytics from the data store and from fast data streams. Adjacency lists are used for old graph snapshots, and circular edge logs are used for incoming updates; the adjacency store is updated from the edge log after crossing an archiving threshold. GraphFly [27] proposes D-trees based on elimination trees, for quick detection of independent graph updates, identification of dependency flows which reduces redundant memory accesses for streaming graphs. Aspen [29] is a graph streaming framework that extends Ligra [37] interface to provide dynamic graph updates. It builds upon their proposed C-Trees functional data structure to provide for fast graph update/query operations for the CPU architecture. Terrace [28] is a CPU-based framework providing a hybrid storage approach for handling skewness in streaming graphs: sorted arrays for low-degree vertices, PMA for medium-degree vertices, and B-trees for vertices with large degrees.

*Dynamic Graph Data Structures on GPU*: The cuSTINGER [5] data structure uses structure-of-arrays (SoA) representation for maintaining edges and large over-provisioned arrays for maintaining the vertex adjacency lists. The SoA representation helps improve coalesced memory accesses. Hornet [4] maintains several block arrays. Each block has a fixed size of a power of two. A vertex maintains its adjacency list within one such fitting block. On insertion, if the allocated block cannot accommodate the new edges, the adjacency list is migrated to a larger block in another block array. The GPMA [8] extends PMA for GPU. The GPMA data structure suffers from uncoalesced memory accesses, overheads in obtaining locks, and lower parallelism if threads conflict on the same segment. These issues get exacerbated especially on real-world graphs with a power-law distribution. The limitation of GPMA are addressed with GPMA+ [8]. LPMA [31] overcomes the array expansion problem of GPMA+ by using a leveled array for maintaining the dynamic graph updates. The aimGraph [32] data structure mainly focuses on the memory management for handling updates for a dynamic graph. By allocating a single large block of global memory, aimGraph eliminates round trips between the CPU and the GPU for memory allocations. Like aimGraph, faimGraph [3] utilizes a memory manager to handle allocation requests entirely on the GPU. When the edge data contains a single value, SoA representation is used, while AoS (array of structures) representation is used when the edge data comprises of several fields.

*Dynamic Graph Frameworks on GPU*: EGraph [33] is a CPU-GPU-based framework for applying the same algorithm on a sequence of different snapshots of the same graph. The framework avoids redundancies in full graph processing by observing that some graph partitions are identical for a sequence of snapshots (spatial similarity), and are likely to be processed again in a short duration (temporal similarity), and proposes a new Loading-Processing-Switching execution model for exploiting these graph snapshot similarities, ensuring workload balance between the GPU SMs, and reduce data transfers between CPU and GPU.

*Dynamic Graph Algorithms*: ConnectIt [38] implements incremental WCC for multi-core CPUs. GConn [21] extends ConnectIt for GPUs. A few other works have addressed challenges in incremental WCC on GPUs [39–41]. Dynamic SSSP, BFS, and MST algorithms are programmed using diff-CSR data structure [1]. A detailed study on dynamic graph algorithms in available in [42]. The computational complexity of sequential dynamic graph algorithms is explored in Ramalingam and Reps [43].

## 8 Conclusion and Future Work

We presented `Meerkat`, a framework for dynamic graph algorithms on GPUs. It builds upon and significantly enhances a hash-based SlabHash data structure. `Meerkat` offers a memory-efficient alternative, proposes new iterators, and optimizes their processing to improve on both the execution time as well as the memory requirement. These enhancements allow dynamic graph algorithms, containing both incremental and decremental updates, to be implemented efficiently on GPUs. We illustrated the effectiveness of the framework using fundamental graph algorithms such as BFS, SSSP, TC, and WCC. As part of future work, we would like to implement more complex graph algorithms using our framework, and also check for the feasibility of approximations to reduce the memory requirement of `Meerkat` further.

## Appendix A Iteration Schemes

We describe two different schemes for enumerating neighbours of a vertex in `Meerkat` object.

### A.1 IteratorScheme1

Algorithm 10 describes the first iteration scheme, namely *IterationScheme1*. The CUDA kernel that uses *IterationScheme1* accepts a dynamic graph object $G$, and

**Algorithm 10:** Iteration Scheme 1 (using `SlabIterator`)

```
1  device function int warpdequeue (bool *to_process) {
2      int work_queue = __ballot_sync(0xFFFFFFFF, *to_process);
3      index = __ffs(work_queue) - 1;
4      if (lane_id() == index) then
5          *to_process = false;
6      end if
7      return index;
8  }
9  function IterationScheme1 (Graph G, Vertex V[vertex_n]) {
10     Vertex_Dictionary* vert_adjs[] = G.get_vertex_adjacencies();
11     if ((thread_id() − lane_id()) < vertex_n)) then
12         bool to_process = (thread_id() < vertex_n);
13         int dequeue_lane = 0; /* queue size is warpsize (i.e 32)        */
           /* warpdequeue() API internally uses __ballot_sync() and
              __ffs warp primitive                                      */
14         while ((dequeue_lane = warpdequeue(&to_process)) ≠ −1) do
15             int common_tid = (thread_id() - lane_id() + dequeue_lane);
16             Vertex src = V[common_tid]; /* all warp threads process
                  neighbours of vertex src                              */
17             SlabIterator iter = G.vert_adjs[src].begin();
18             SlabIterator last = G.vert_adjs[src].end();
               /* warp cooperative processing of adjacency slabs of vertex
                  src                                                   */
19             while (iter ≠ last) do
20                 Vertex v = iter.get_pointer(lane_id()); /* each warp thread index to
                      different slab entry                              */
21                 if (is_valid_vertex(v)) then
                       /* Process adjacent vertex, if the slab-entry is not
                          TOMBSTONE_KEY                                 */
22                 end if
23                 ++iter;
24             end while
               /* Post-processing                                       */
25         end while
26     end if
27 }
```

an array of vertices $A$ of size $vertex\_n$, whose adjacencies in the graph $G$ are to be visited. For example, this array $A$ could be holding a frontier of vertices in the $BFS$ algorithm whose adjacencies have to be visited in a given iteration. The CUDA kernel is invoked with $t$ threads where, $t = \left\lfloor \frac{vertex\_n+BS-1}{BS} \right\rfloor$. `BS` refers to the threads-per-block chosen for the CUDA kernel and it must be a multiple of the warp size. In other words, the kernel is invoked with a number of threads equal to the smallest multiple of the thread-block size above or equal to $vertex\_n$.

The thread-block size must be a multiple of the warp size for the successful execution of intra-warp communication primitives such as `__ballot_sync` [9], used for work-cooperative work strategy extensively used in algorithms programmed using `Meerkat`. The expression $(thread\_id() - lane\_id())$ finds the thread-id of the first thread in a warp (See Table 4). The predicate at Line 11 allows only those warps which have at least one thread with thread-ids less than the number of vertices in the graph, to proceed with the computation.

At line 12, we identify those threads whose thread-ids are less than *vertex_n* and can validly index into $V$, the array storing the list of vertices to process.

The `warpdequeue()` function (see lines 1–7) identifies those threads within the warp having a vertex remaining to be processed and stores in the variable *work_queue* (see line 2). Each set bit in the work queue corresponds to one unique thread within the warp that needs to be processed with the value of variable $to\_process$ equals to `true`. Using the CUDA function `__ffs()` (find first set-bit), we elect the first outstanding thread from *work_queue* and store it in the local variable *index* (see line 3). The first outstanding bit is the first set bit starting from the least significant bit position. If all the bits in the variable *work_queue* have a value of zero, then the variable *index* will get a value of -1. The local variable *to_process* passed by reference to the `warpdequeue()` function is set to `false` for the warp thread at *lane_id index*. The `warpdequeue()` function then returns the value of the variable *index* (see line 7).

The value returned by the `warpdequeue()` function is assigned to the variable *dequeue_lane* (see line 14). The `while` loop terminates when the value returned by the `warpdequeue()` function is -1. Thus the loop at Lines 14–25 continues as long as there is an outstanding thread within the warp whose associated vertex is left to process. All the threads within the warp index into the same position of the `Vertex` array $V$, and the `Vertex` variable *src* will have the same value for all threads in the warp (see lines 15–16). A pair of `SlabIterators`, namely `iter`, and `last`, are constructed (lines 17–18) to traverse through the slabs storing the adjacent vertices of the `Vertex` $src$ (within the loop at lines 19–23). All the threads within the warp perform a coalesced memory access to the contents of the slab represented by `iter` (see line 20). If the value fetched by the thread from the current slab represents a valid vertex, (see line 21), the thread processes it as the adjacent vertex. After processing the current slab, the iterator `iter` is incremented (see line 23) so that it refers to the next slab in the sequence.

## A.2 IteratorScheme2

The Algorithm 11 presents *IterationScheme2*. Unlike *IterationScheme1* which uses `SlabIterators`, *IterationScheme2* uses `BucketIterators` and eliminates the use of a work queue of vertices, and instead operates with a grid-stride loop [44]. This iteration scheme does not restrict the number of thread blocks. However, for the warp-level primitives (such as `__shfl_sync`) to work correctly on the slabs, the number of active threads within a thread block must be a multiple of the warp size. Since the adjacencies of a vertex are distributed among multiple slab-lists, a slab-list can thus be identified with a $\langle v, i\rangle$ pair, which refers to the $i^{th}$ slab-list of a vertex $v$. Such pairs are stored in the `slab_list_vertex` and `slab_list_index` device vectors. Each loop iteration (in lines 6–17) within a warp traverses and processes all the slabs contained in one slab-list uniquely identified by its $\langle v, i\rangle$ pair. The $\langle v, i\rangle$ pairs are represented in the `slab_list_vertex` and `slab_list_index` device vectors. For example, if a vertex frontier contains two vertices $v_i$ and $v_j$, con-

**Algorithm 11:** Iteration Scheme 2 (using `BucketIterator`)

```
1  function IterationScheme2 (Graph G, Vertex slab_list_vertex[n], int slab_list_index[n]) {
2      Vertex_Dictionary *vert_adjs= &(G.Vert_Dict[0])
3      int warps_n = (blockDim.x * gridDim.x) / warp_size()
4      int global_warp_id = thread_id() / warp_size()
5      int i = global_warp_id
6      while ($i < n$) do
7          Vertex src = slab_list_vertex[i]
8          int index = slab_list_index[i]
           /* Process source vertex                                              */
9          BucketIterator iter = vert_adjs[src].begin_at(index)
10         BucketIterator last = vert_adjs[src].end_at(index)
           /* Iterate over adjacent vertices                                     */
11         while ($iter \neq last$) do
               /* each thread in the warp fetches different vertex from
                  based on its lane-id.                                          */
12             Vertex v = *iter.get_pointer(lane_id())
13             if ($is\_valid\_vertex(v)$) then
                   /* Process adjacent vertex                                    */
14             end if
15             ++iter
16         end while
           /* Post-processing                                                    */
17         i += warps_n
18     end while
19 }
```

taining 3 and 2 slablists respectively. To enable *IterationScheme2* to traverse through all the their respective slabs, $slab\_list\_vertex$ [] is initialized as $[v_i, v_i, v_i, v_j, v_j]$, and $slab\_list\_index$ [] is initialized as [0, 1, 2, 0, 1].

The total number of warps in the kernel is computed and stored in *warps_n* (at line 3). Each warp is uniquely identified with a global warp id (computed at line 4). By using its *global_warp_id* as the initial value for index variable $i$, each warp identifies its slab list $\langle v, i \rangle$ to process by indexing into the `slab_list_vertex` and `slab_list_index` vectors (see lines 7–8). This index is incremented at the stride of the total number of warps in the grid for the CUDA kernel (line 17). CUDA kernel can be called with a total number of threads being lesser than the total number of slab lists for the input graph object.

## Appendix B Decremental SSSP - Helper Functions for SSSP Distance Invalidation and its Propagation

**Algorithm 12:** Decremental SSSP - Invalidate Distance Kernel

```
device function Invalidate (Edges batch_edges[N], tree_node D[vertex_n]) {
    uint t = thread_id()
    while ($t < N$) do
        Vertex dst = batch_edges[t].dst
        /* Invalidate tree node for vertex dst if ($parent_{dst}$, dst) is a
           batch edge for deletion */
        if ($D[dst].parent == batch\_edges[t].src$) then
            D[dst] = ⟨INF, INVALID⟩
        end if
        t += threads_n()
    end while
}
```

**Algorithm 13:** Decremental SSSP - Propogate Invalidation Kernel

```
device function PropogateInvalidation (tree_node D[vertex_n], Vertex src) {
    uint i = threads_id()
    /* Grid-stride loop: each thread checks if the vertex i is
       reachable to source src in the dependence tree */
    while ($i < vertex\_n$) do
        if ($D[i] \neq \langle INF, INVALID \rangle$) then
            Vertex ancestor = D[i].parent
            while (ancestor != src) do
                /* Traverses to the source vertex src: loops until src or
                   invalidated vertex is found in path */
                tree_node $d_a$ = D[ancestor]
                if ($d_a == \langle INF, INVALID \rangle$) then
                    /* Invalidate vertex i if ancestor a is invalidated */
                    D[i] = ⟨INF, INVALID⟩
                    break
                end if
                ancestor = D[$d_a$].parent
            end while
        end if
        i += threads_n()
    end while
}
```

**Author Contributions** Unnikrishnan C and Rupesh Nasre conceived the presented idea, the goals and aims. Kevin Concessao and Ricky Dev were responsible for the design and implementation of the computer source code, testing; collection, validation, and intepretation of results. All authors have contributed to the initial draft of the manuscript. Unnikrishnan C and Rupesh Nasre provided critical feedback, shaped the analysis, reviewed and edited the manuscript. Unnikrishnan C and Rupesh Nasre have helped supervise the project.

**Funding** This work is funded by National Super Computing Mission, India. Project Number: DST/NSM/R&D_Exascale/2021/18.

**Data Availability** No datasets were generated or analysed during the current study.

**Code Availability** The source code is available at https://github.com/meerkat-pkd/Meerkat.git.

## Declarations

**Conflict of interest** The authors have no conflict of interest to declare that are relevant to the content of this article.

**Ethical Approval** Not applicable.

**Consent for Publication** All authors have consented to the publication of this manuscript.

## References

1. Malhotra, G., Chappidi, H., Nasre, R. Rauchwerger, L.: (ed.) Fast Dynamic Graph Algorithms. Languages and Compilers for Parallel Computing - 30th International Workshop, LCPC 2017, College Station, TX, USA, October 11-13, 2017, Revised Selected Papers, Vol. 11403 of Lecture Notes in Computer Science, 262–277 (Springer), (2017). https://doi.org/10.1007/978-3-030-35225-7_17
2. Awad, M.A., Ashkiani, S., Porumbescu, S.D., Owens, J.D.: Dynamic graphs on the GPU (2020). https://doi.org/10.1109/IPDPS47924.2020.00081
3. Winter, M., Mlakar, D., Zayer, R., Seidel, H.-P., Steinberger, M.: Faimgraph: high performance management of fully-dynamic graphs under tight memory constraints on the gpu. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18 (IEEE Press), (2018)
4. Busato, F., Green, O., Bombieri, N., Bader, D. A.: Hornet: an efficient data structure for dynamic sparse graphs and matrices on gpus. In: 2018 IEEE High Performance extreme Computing Conference (HPEC), 1–7 (IEEE), (2018). https://doi.org/10.1109/HPEC.2018.8547541
5. Green, O., Bader, D. A.: cuSTINGER: supporting dynamic graph algorithms for gpus. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), 1–6 (2016). https://doi.org/10.1109/HPEC.2016.7761622
6. Ashkiani, S., Farach-Colton, M., Owens, J. D.: A dynamic hash table for the GPU. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 419–429 (2018). https://doi.org/10.1109/IPDPS.2018.00052
7. Ediger, D., McColl, R., Riedy, J., Bader, D. A.: Stinger: high performance data structure for streaming graphs. In: 2012 IEEE Conference on High Performance Extreme Computing, 1–5 (2012). https://doi.org/10.1109/HPEC.2012.6408680
8. Sha, M., Li, Y., He, B., Tan, K.-L.: Accelerating dynamic graph analytics on GPUs. Proc. VLDB Endow. **11**, 107–120 (2017). https://doi.org/10.14778/3151113.3151122 https://doi.org/10.14778/3151113.3151122
9. Nvidia. Nvidia warp primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/
10. CUDA C++ Programming Guide - Cooperative Groups (2023). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups. [Accessed 08-11-2023]
11. Vora, K., Gupta, R., Xu, G.: Kickstarter: fast and accurate computations on streaming graphs via trimmed approximations. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, 237–251 (Association for Computing Machinery, New York, NY, USA), (2017). https://doi.org/10.1145/3037697.3037748
12. Arasu, A., Novak, J., Tomkins, A., Tomlin, J. A.: Pagerank computation and the structure of the web: experiments and algorithms (2002)

13. Makkar, D., Bader, D. A., Green, O.: Exact and parallel triangle counting in dynamic graphs. In: 2017 IEEE 24th International Conference on High Performance Computing (HiPC), 2–12 (2017). https://doi.org/10.1109/HiPC.2017.00011
14. De Domenico, M., Lima, A., Mougel, P., Musolesi, M.: The anatomy of a scientific rumor. Scientific reports **3**(1), 2980 (2013). https://doi.org/10.1038/srep02980
15. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: membership, growth, and evolution. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06, 44–54 (Association for Computing Machinery, New York, NY, USA), (2006). https://doi.org/10.1145/1150402.1150412
16. Takac, L., Zábovský, M.: Data analysis in public social networks. In: International Scientific Conference and International Workshop Present Day Trends of Innovations 1–6 (2012)
17. Leskovec, J., Lang, K. J., Dasgupta, A., Mahoney, M. W.: Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters (2008). arXiv:0810.1355
18. Leskovec, J., Huttenlocher, D., Kleinberg, J.: Signed networks in social media. CHI '10, 1361–1370 (Association for Computing Machinery, New York, NY, USA, 2010). https://doi.org/10.1145/1753326.1753532
19. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. In: Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, MDS '12 (Association for Computing Machinery, New York, NY, USA), (2012). https://doi.org/10.1145/2350190.2350193
20. DIMACS Implementation Challege. http://www.diag.uniroma1.it//challenge9/download.shtml
21. Hong, C., Dhulipala, L., Shun, J.: Exploring the design space of static and incremental graph connectivity algorithms on gpus. Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, PACT '20, 55–69 (Association for Computing Machinery, New York, NY, USA), (2020). https://doi.org/10.1145/3410463.3414657
22. Zhu, X.: Livegraph: a transactional graph storage system with purely sequential adjacency list scans. Proc. VLDB Endow **13**, 1020–1034 (2020). https://doi.org/10.14778/3384345.3384351
23. Jaiyeoba, W., Skadron, K.: Graphtinker: a high performance data structure for dynamic graph processing. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 1030–1041 (2019). https://doi.org/10.1109/IPDPS.2019.00110
24. Kumar, P., Huang, H.H.: Graphone: a data store for real-time analytics on evolving graphs. ACM Trans. Storage **15**(4), 1–40 (2020). https://doi.org/10.1145/3364180
25. De Leo, D., Boncz, P.: Teseo and the analysis of structural dynamic graphs. Proc. VLDB Endow. **14**, 1053–1066 (2021). https://doi.org/10.14778/3447689.3447708
26. Fuchs, P., Margan, D., Giceva, J.: Sortledton: a universal, transactional graph data structure. Proc. VLDB Endow. **15**, 1173–1186 (2022). https://doi.org/10.14778/3514061.3514065
27. Chen, D., et al.: Graphfly: efficient asynchronous streaming graphs processing via dependency-flow. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '22 (IEEE Press, 2022)
28. Pandey, P., Wheatman, B., Xu, H., Buluc, A.: Terrace: a hierarchical graph container for skewed dynamic graphs. In: Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, 1372–1385 (Association for Computing Machinery, New York, NY, USA), (2021). https://doi.org/10.1145/3448016.3457313
29. Dhulipala, L., Blelloch, G. E., Shun, J.: Low-latency graph streaming using compressed purely-functional trees. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, 918–934 (Association for Computing Machinery, New York, NY, USA), (2019). https://doi.org/10.1145/3314221.3314598
30. Afarin, M., Gao, C., Rahman, S., Abu-Ghazaleh, N., Gupta, R.: Commongraph: graph analytics on evolving data. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, 133–145 (Association for Computing Machinery, New York, NY, USA), (2023). https://doi.org/10.1145/3575693.3575713
31. Zhang, F., Zou, L., Yu, Y. Zhang, W., Zou, L., Maamar, Z., Chen, L.: (eds) Lpma - an efficient data structure for dynamic graph on gpus. Web Information Systems Engineering – WISE 2021, 469–484 (Springer International Publishing, Cham), (2021)
32. Winter, M., Zayer, R., Steinberger, M.: Autonomous, independent management of dynamic graphs on gpus. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), 1–7 (2017). https://doi.org/10.1109/HPEC.2017.8091058

33. Zhang, Y., et al.: Egraph: efficient concurrent GPU-based dynamic graph processing. IEEE Transactions on Knowledge and Data Engineering **35**, 5823–5836 (2023). https://doi.org/10.1109/TKDE.2022.3171588
34. Vora, K., Koduru, S. C., Gupta, R.: Aspire: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, 861–878 (Association for Computing Machinery, New York, NY, USA), (2014). https://doi.org/10.1145/2660193.2660227
35. Bender, M. A., Hu, H.: An adaptive packed-memory array. In: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '06, 20–29 (Association for Computing Machinery, New York, NY, USA), (2006). https://doi.org/10.1145/1142351.1142355
36. Fagerberg, R.: Cache-Oblivious B-Tree, 121–123 (Springer US, Boston, MA), (2008). https://doi.org/10.1007/978-0-387-30162-4_61
37. Shun, J., Blelloch, G. E.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, 135–146 (Association for Computing Machinery, New York, NY, USA), (2013)https://doi.org/10.1145/2442516.2442530
38. Dhulipala, L., Hong, C., Shun, J.: Connectit: a framework for static and incremental parallel graph connectivity algorithms. Proc. VLDB Endow. **14**, 653–667 (2020). https://doi.org/10.14778/3436905.3436923
39. Soman, J., Kishore, K., Narayanan, P. J.: A fast gpu algorithm for graph connectivity. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 1–8 (2010). https://doi.org/10.1109/IPDPSW.2010.5470817
40. Jaiganesh, J., Burtscher, M.: A high-performance connected components implementation for gpus. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '18, 92–104 (Association for Computing Machinery, New York, NY, USA), (2018). https://doi.org/10.1145/3208040.3208041
41. Sutton, M., Ben-Nun, T., Barak, A.: Optimizing parallel graph connectivity computation via subgraph sampling. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 12–21 (2018). https://doi.org/10.1109/IPDPS.2018.00012
42. Cheramangalath, U., Nasre, R., Srikant, Y.N.: Dynamic Graph Algorithms, pp. 137–151. Springer International Publishing, Berlin (2020). https://doi.org/10.1007/978-3-030-41886-1_6
43. Ramalingam, G., Reps, T.: On the computational complexity of dynamic graph problems. Theor. Comput. Sci. **158**, 233–277 (1996). https://doi.org/10.1016/0304-3975(95)00079-8
44. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops | NVIDIA Technical Blog — developer.nvidia.com. https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/. [Accessed 08-11-2023]