Towards Sufficient GPU-accelerated Dynamic Graph Management: Survey and Experiment

Yinnian Lin
Peking University
Bejing, China
linyinnian@pku.edu.cn

Lei Zou
Peking University
Bejing, China
zoulei@pku.edu.cn

Xunbin Su
Peking University
Bejing, China
suxunbin@pku.edu.cn

ABSTRACT

Dynamic graph management (DGM) systems are designed to effectively handle changing graph data, which is a fundamental problem for many graph-based applications. Recently, researchers have designed GPU-based solutions for DGM and its downstream applications, thanks to GPUs' massive parallelism power. However, there is a lack of universal models that summarize the features and design principles of GPU-accelerated DGM systems. Additionally, existing studies test GPU-based DGM systems without unified metrics and workloads. Under this circumstance, we propose a conceptual model for GPU-accelerated DGM to demonstrate a DGM system's components, key primitives, and optimization choices. Next, we evaluate six representative systems, testing their update and query performance with unified metrics and workloads of different algorithmic behaviors. We also extend existing systems to seek insight to fill the current research gap in multi-GPU support, concurrency control, resource utilization, and so on. Our evaluation yielded new insights on the pros and cons of different systems: (1) Hashing-based systems perform best for graph updates but may not be suitable for all applications. (2) Finding a system that fits all workloads is challenging, and hybrid data storage may be a solution. (3) To select the most suitable DGM system for a specific workload, it is essential to consider hardware-related metrics. Finally, we provide recommendations and suggestions for future studies based on our experimental results and observations.

PVLDB Reference Format:
Yinnian Lin, Lei Zou, and Xunbin Su. Towards Sufficient GPU-accelerated Dynamic Graph Management: Survey and Experiment. PVLDB, 18(3): 599 - 612, 2025.
doi:10.14778/3712221.3712228

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/pkumod/GPU_DGM.

1 INTRODUCTION

In recent years, the analysis of massive graphs, which are powerful representations of entities and their interconnections, has become increasingly important in academic and industrial domains. This field encompasses research and applications in areas such as the Internet of Things [2], social network analysis [22], and protein property prediction [36]. However, the management of graph data remains a challenge for two reasons: Firstly, the scale of graph data can be immense, often containing tens of millions of edges or more [11, 39]. Secondly, graphs are typically dynamic, with edges and vertices undergoing frequent and rapid updates. For instance, X processes over 500 million new posts and numerous user interactions daily [3].

To address the challenges associated with massive and dynamic graph data [41, 48, 59], various Dynamic Graph Management (DGM) frameworks have emerged, including Aspen [18], Livegraph [70], Sortledton [25], Terrace [50], and Teseo [44]. These systems differentiate themselves from static frameworks by their ability to perform analytical algorithms such as Breadth-First Search (BFS), Single Source Shortest Path (SSSP), and PageRank (PR), while concurrently handling graph updates with accuracy and efficiency. Nevertheless, DGM systems continue encountering difficulties in managing large data volumes and high update velocities, which calls for improvement in dynamic data storage and parallel updates. Furthermore, leveraging hardware-related optimizations, such as cache utilization, I/O patterns, and workload balancing, is instrumental in performance enhancement.

Graphics Processing Units (GPUs) have experienced a substantial increase in computing power and accessibility, making them a viable option for addressing the challenges of DGM [58]. Performance improvements are unsustainable due to the physical and power limitations of manufacturing as transistor growth plateaus. However, the expansion of data volume continues to surpass the advancement in hardware, highlighting the importance of parallel computing as a potential resolution to this dilemma. Unlike CPUs, GPUs allocate more on-chip resources to computation rather than cache management and flow control. A modern GPU contains thousands of cores that, although less powerful individually, work together through a Single Instruction Multiple Thread (SIMT) architecture. The achievement of parallelism is crucial in graph analytics, particularly for managing large-scale data, as many graph algorithms entail simple, yet independent operations such as tagging, filtering, and intersection across numerous vertices or edges [49]. Therefore, existing graph libraries are either augmented with GPU-based extensions or specifically crafted for GPU environments [13, 14, 51, 62, 67] to exploit GPUs for enhanced throughput.

Challenges. However, managing large-scale and dynamically evolving graph data on GPUs remains challenging, with the following issues prevalent:

First, optimizing GPU performance requires in-depth knowledge of workload characteristics such as memory access patterns. However, DGM operations and graph algorithms frequently involve

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712228

unpredictable memory allocations and random accesses to neighbor lists. For instance, the continuous insertion of edges may trigger the extension of data structures that disrupt GPU processing. Additionally, irregular memory access patterns emerge when algorithms enumerate vertices through their neighbors. The prediction of the next data access location becomes difficult due to the low spatial and temporal locality inherent in analytic workloads. As a result, many existing systems opt to pre-allocate a significant amount of memory, organizing it into fixed-sized blocks or pages. Moreover, various strategies for neighbor list storage have been proposed to facilitate rapid access to adjacent vertices. Nevertheless, optimizations that improve locality in static graphs may not be effective in DGM contexts. For example, vertex ordering, a common technique to enhance the locality of graph algorithms [32, 34, 63], incurs additional overhead with ongoing graph updates.

Second, maintaining a balance between update performance and query efficiency for DGM on GPUs presents a significant challenge. For instance, utilizing a well-constructed hash table for neighbor lists allows for the updating or the existence verification for edges in approximately constant time. Still, this approach sacrifices the efficiency of list enumeration due to the overhead of preserving extra space and keeping unordered elements. Conversely, although a continuously stored sorted array improves enumeration efficiency, inserting new elements is expensive, triggering extra data movement. Consequently, when designing DGM systems, considering the trade-off between update and query performance as workloads vary is important. This experimental study offers insights and recommendations regarding the most appropriate designs for different applications.

Third, the design of a fully optimized DGM framework should comply with the inherent characteristics of GPUs. Mere transplantation of CPU-based systems onto GPUs is insufficient for achieving optimal performance. Instead, GPU-specific optimizations are essential. For instance, branch statements may result in warp divergence, a phenomenon where certain threads become idle because all threads within a warp must execute the same instruction simultaneously. Moreover, skewed data graphs can exacerbate issues of memory locality. Shared memory, a user-visible, high-speed, and configurable on-chip memory resource, can act as a user-defined cache to minimize off-chip memory accesses. Additional critical factors include memory access coalescing, managing out-of-memory situations, and achieving workload balance.

Contributions. To address the identified challenges, we conducted an extensive experimental study on GPU-based DGM systems, focusing on their design choices and performance across various workloads. Our contributions are as follows:

- We conducted an evaluation of six state-of-the-art GPU-based DGM systems: cuSTINGER [29], Hornet [9], faimGraph [65], Gunrock [62], GPMA [56], and LPMA [71]. We assessed the systems' performance metrics, including update latency, query throughput, and resource utilization. Additionally, we evaluated the system's proficiency across different analytical workloads, utilizing datasets that ranged in scale.
- To facilitate a comprehensive understanding, we introduce a universal model for GPU-based DGM, which provides a structured overview of GPU-based DGM systems by delineating them into three distinct dimensions: system components, critical operations, and design decisions. To our knowledge, this work is the first attempt to propose a coherent conceptual model for GPU-based DGM.
- We identify the most appropriate GPU-based DGM solution for various applications. Our experiments substantiate previous experimental results, extend new observations, and discover possible future directions.

2 PRELIMINARIES

2.1 Problem definition

We first briefly review the terminology we use to define the DGM problem.

A graph G is a tuple $G = \{V, E\}$, where $V(G)$ is a set of vertices, $E(G) \subset V(G) \times V(G)$ is an edge set. A dynamic graph is defined by an initial graph $G = (V, E)$, and an infinite series of operation batches $O = \{\Delta G_0, \Delta G_1, ...\Delta G_t ...\}$, where each batch consists of two kinds of operations: insertion and deletion. t denotes a timestamp. The batches are ordered by their issued timestamps.

In this study, we define graph updates as edge insertions and deletions, as vertex insertion/deletion can be implemented through edge operations. We prioritize batch updates over real-time processing, consistent with existing DGM systems on GPUs[9, 29, 56, 62, 71]. During batch execution, updates arrive continuously and are processed once a full batch is accumulated, without overlap between batches. This ensures sufficient data to leverage GPUs' parallel processing capability and optimize bandwidth between GPU memory and computational kernels.

Table 1: Frequently Used Notations

Notation	Description
$G(V, E)$	A graph consisting of vertices V and edges E
ΔG	A batch of updates on G
$V(G)$	The set of vertices in G
$E(G)$	The set of edges in G
$d(v)$	The degree of a vertex v
$N(v)$	The neighbor list of vertex v
$\overline{d}$	Average degree

We categorize the fundamental workloads of DGM into two types: updates and queries. An update alters the data graph by adding or removing edges. A query executes computations on the graph and yields results, ranging from simple, such as verifying the presence of an edge, to complex, like identifying frequent patterns. Consequently, with ongoing edge insertions and deletions, the results of queries keep changing.

2.2 Basic graph structure

This subsection introduces three primary graph data structures: the adjacency matrix, adjacency list, and Compressed Sparse Row (CSR) format. These are the most prevalent layouts for static graph data and are widely adopted by existing systems to handle dynamic graphs.

2.2.1 Adjacency matrix. An adjacency matrix, $M \in \mathbb{R}^{|V|\times|V|}$, is a data structure that captures the edges of a graph, where $M(u, v)$ signifies the presence or records the weight of the edge between u and v. Updating or querying an edge can be done in constant time on average with an adjacency matrix. A significant drawback, however, is the quadratic space complexity $O(|V|^2)$ which leads to high memory costs. For large graphs with a power-law degree distribution, the adjacency matrix tends to be sparse, as most vertices have a small number of neighbors. This sparsity results in a memory inefficiency issue because we must still allocate space for numerous zero entries in a dense matrix. One solution is using sparse matrix representations, which store only the non-zero elements and their indices. Nevertheless, computations involving sparse matrices can be more complex due to the irregular data layouts they introduce [35].

2.2.2 Adjacency list. An adjacency list consists of $|V|$ neighbor lists and structures to index these lists (e.g., a pointer array). Each vertex $u \in V$ has a neighbor list $N(u)$ implemented as an array, a hash table, or a tree-based structure. These neighbor lists are often sorted to facilitate edge searches and set intersections. However, maintaining sorted lists using arrays can incur overhead during insert operations. When inserting a new edge, one must locate the appropriate position for the new entry and potentially reorganize the list by shifting elements. Therefore, existing systems reserve extra space within data structures to accommodate new insertions and develop strategies to dynamically manage reserved space during runtime according to current and former workloads.

2.2.3 Compressed Sparse Row(CSR). CSR concatenates neighbor lists into a contiguous edge array. An index of offsets, denoted as $Offset$, records the beginning position of each list, allowing for quick location of a required neighbor list. For example, to locate $N(v_o)$, we should read $Offset[0]$ and $Offset[1]$. $Offset[0]$ denotes the position of the first element of $N(v_o)$ in the edge array, and $offset[1] - offset[0]$ indicates the length of $N(v_o)$. CSR introduces less random memory access, making it a popular choice for static graph processing. However, since CSR is compact, updates lead to heavy overhead in moving data in the edge list and modifying $Offset$.

2.3 Related Work

CPU-based DGM: Various DGM systems have been proposed on CPUs, in which CSR is a popular choice as a basic representation. A common idea is dividing neighbor lists into fixed-size blocks for ease of memory management and future insertion [21, 24, 46, 66]. Livegraph [70] suggests a Transactional Edge Log and corresponding concurrency control mechanism for efficient list scanning. Teseo [44] balances the sparse neighbor array using a fat tree. GraphIn [55], GraphOne [43], and Evograph [54] buffer the graph update in an edge list and conduct batch processing.

Adjacency-list-based structures focus on designing data layout for a neighbor list [1, 10, 16, 17, 19, 27, 37, 38, 47, 61]. For instance, GraPU [57] utilizes reorganization techniques in the update buffer to minimize conflicts during concurrent updates. Risgraph [23] proposes Indexed Adjacency Lists using sparse arrays to improve inter-update parallelism. Terrace [50] applies appropriate structures for vertices of different degrees. Sortledton [25] uses an unrolled skip list to store blocks of edges.

CPU-based DGM v.s. GPU-based DGM: Among the various existing DGM systems, GPUs are the most preferred hardware due to their massive parallelism power [9, 29, 56, 62, 65, 71]. Migrating state-of-the-art CPU-based systems to a GPU environment is challenging due to inherent differences between the two types of hardware. Firstly, many of the leading CPU-based systems rely on tree-based structures that involve pointer jumping, which results in a random memory access pattern. This pattern is less efficient on GPUs and does not leverage the benefits of GPUs' coalesced memory access. Secondly, GPU-based DGM research has not adequately addressed issues prominent in CPU environments, such as isolation, consistency, contention control, and freshness. Most existing GPU-based DGM systems prioritize overall throughput and typically support update-only or query-only batches. Furthermore, in the current paradigm, multiple batches do not overlap, leading to systems that pre-eliminate duplicate updates in advance. In essence, current research efforts have made use of parallelism and throughput advantages of GPUs in idealized scenarios to some extent, often disregarding the complexities of isolation, consistency, and freshness. However, these factors remain crucial for developing GPU-based DGM systems suitable for real-world applications. Although it is beyond the scope of this paper to explore the implementation of consistency and contention control for GPU-based DGM, it is an open and important problem that needs further investigation.

Two recent surveys [12, 26] have analyzed and evaluated GPU-based DGM systems and databases. Our paper offers unique and complementary insights as well as experimental findings that stand apart from these prior studies. Cao et al. [12] conducted a performance and resource utilization analysis of current GPU database systems, delivering micro-architectural insights that are missing in previous research. Their findings reveal that many queries in GPU-based databases do not fully utilize available GPU resources. This has guided our use of the roofline model to identify performance bottlenecks, filling a gap in the existing literature on GPU-based DGM systems. Gao et al. [26] provided a detailed overview of various systems and discussed their design distinctions. In addition, we introduce a conceptual model for GPU-based DGM and evaluate existing systems within this model, offering a novel and high-level perspective. Additionally, while [26] is a technical survey, our contribution is an experimental effort that presents extensive results to support our observations and offer insights for future directions.

Motivated by the literature review above, we extract a conceptual model that discusses components, primitives, and optimization chances of existing GPU-based DGM systems in Section 3, followed by a detailed discussion of these systems in our model in Section 4.

3 MODELING GPU-BASED DGM

To thoroughly evaluate current GPU-based DGM systems under a wide range of workloads, it is imperative to develop a universal model that identifies and establishes common abstractions across them. This section introduces a conceptual model that encapsulates the shared characteristics of GPU-based DGM systems. This model includes system components, basic primitives, and prevalent design

decisions. We also explore various strategies and considerations for performance optimization. Our research represents the first effort to develop a universal model for analyzing and understanding existing GPU-based DGM systems.

3.1 System components and primitives

3.1.1 System components. Fig. 1 shows common components of existing GPU-based DGM systems, including a CPU host, GPU kernels, topology structure, attribute storage, and optional auxiliary structures that speed up specific operations. Fig. 1 also demonstrates interactions between these components and how they adapt to GPU memory hierarchy and multi-GPU scenarios.

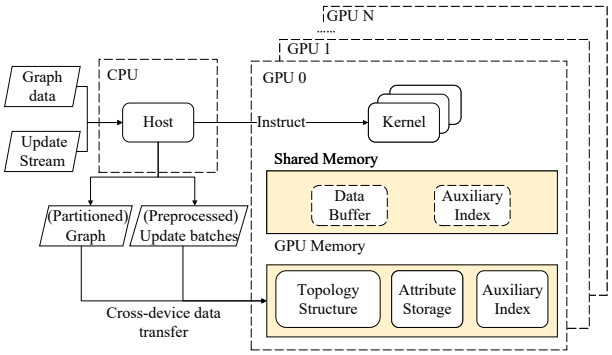


Figure 1: A universal model for GPU-based DGM systems. Solid rounded rectangles indicate necessary components and dashed ones represent optional structures.

CPU host: A typical DGM system is controlled by a CPU host that handles system initialization and data pre-processing. A CPU host also manages GPU kernels by issuing or terminating computation tasks. Besides, a CPU host controls memory allocation and data transfer between CPUs and GPUs. This can result in performance loss due to inter-device synchronization and limited cross-device IO bandwidth. In the context of multiple GPUs, an important task for a CPU host is partitioning a data graph and distributing data partitions and batches of updates to kernels running on different GPUs. We highlight graph partitioning because storing an entire copy on every GPU may not be wise, which brings redundant updates and cannot handle data graphs larger than GPU memory. This is also a common concern for GPU-accelerated graph analysis[15, 30, 68].

GPU kernels: GPU kernels perform computations at the behest of the CPU host, and their design must be carefully crafted to avoid low parallelism and inefficient resource utilization. This paper assumes that the GPU is a discrete device, working alongside the CPU. As a result, GPU kernels usually do not have direct access to the machine's main memory. Instead, modern GPUs are equipped with device memory, which has a different architecture than the main memory and features higher parallel access bandwidth. A GPU also has a hierarchical memory layout that includes registers, layers of cache, shared memory, and device memory (a.k.a., GPU memory).

The roles of GPU kernels can vary; some are tasked with updating graph data, while others are responsible for implementing specific algorithms. However, all kernels require some level of access to device memory. Many prevalent graph algorithms are characterized by straightforward computational logic but entail extensive memory requests. Consequently, the efficiency of GPU kernels is frequently constrained by memory bandwidth, making their performance largely IO-bounded. [7].

Topology structure: The topology structure maintains edges and vertices. The neighbor lists, which represent the connections between vertices, can be stored in two primary ways: either as separate adjacency lists for each vertex or concatenated into a continuous space known as Compressed Sparse Row (CSR) format. Another alternative approach is to store edges in a dense matrix, which allows for constant-time edge insertions. However, it is not common in existing GPU-based DGM systems due to its poor memory efficiency. In real-world applications, graphs often contain a large number of edges that are subject to frequent updates. As a result, the topology structure tends to be the most accessed data structure within a DGM system, which can lead to performance bottlenecks, given the high frequency of access and potential contention.

Attribute storage: To run analytical workloads, a DGM system is expected to support different types of attributes. When only simple attributes are needed, such as a numerical weight or a fixed-length label, a DGM system can merge attribute storage with its topology structure. However, according to existing studies [60], using attribute storage independent from topology data is recommended for complex attributes consisting of multiple data. Attribute storage separates the memory footprints of attribute data from a topology structure, thereby reducing costs from updates that trigger expansion or self-balancing where data movement leads to overhead.

In existing GPU-based DGM systems, attribute storage attracts less design effort than a topology structure for two reasons. First, only simple attributes are needed for many commonly used graph analytical workloads like Breath-First-Search (visited flags), Single Source Shortest Path (edge weights), and PageRank (PageRank values and edge weights). In these cases, attributes are usually attached to a topology structure. Second, even when attribute storage independent from a topology layout is necessary, developers can adapt existing GPU-based indexing techniques like key-value stores and hash tables, which are well-studied topics on GPUs [5, 6, 33, 45, 69]. By contrast, optimizations for a topology structure are inadequate because it is more sensitive to data graphs and workloads, hence more challenging.

Auxiliary structures: Auxiliary structures have been introduced to speed up computation and mitigate IO costs. By examining the current literature, we classify these structures into two main categories: (1) Data buffers are designed to store frequently accessed data during computation, reducing the need for costly topology structure access. These buffers can leverage shared memory, which is a fast but limited resource shared among threads within a warp, analogous to caches in CPUs. (2) Data indices retain metadata to facilitate quicker graph updates. For example, faimgraph [65] maintains a queue that tracks data pages available for reuse. Similarly, Hornet [9] utilizes a B+ tree and a bit tree to accelerate identifying available slots for insertions. As the design of these auxiliary structures is highly dependent on specific implementations, a more

in-depth exploration of the auxiliary structures employed in various systems will be provided in Section 4.

3.1.2 Basic primitives. DGM systems provide a comprehensive set of primitive APIs tailored for various downstream tasks. These APIs facilitate operations such as updating and querying graphs, which are succinctly summarized in Table 2. For conciseness, we have excluded straightforward get/set functions, such as those used for retrieving edge weights.

Update: All DGM systems offer edge update primitives that insert, delete, or modify edges. Vertex insertion and deletion can be conducted by inserting or deleting all adjacent edges. The process of edge insertion within DGM systems is known to be a bottleneck and has consequently become a critical area for performance optimization across the board. When inserting an edge, the system must identify an appropriate storage location, verify its availability and validity, and store the data. This can entail searching in neighbor lists, reallocating memory, and sometimes reconstructing a partition or the entire data structure. In contrast, edge deletion can be performed more efficiently through a lazy strategy, which simply marks deleted data as invalid. Current systems handle invalid data by reusing it during new insertions or reclaiming the space through checkpoints and reorganization. As a result, deletion primarily involves locating affected data slots within a data structure.

Query: In the design of DGM systems, query operations are occasionally overlooked to prioritize higher update throughput. For example, some systems maintain unsorted neighbor lists, enabling the efficient appending of incoming neighbors to the list's end. However, this design choice complicates finding a specific neighbor (i.e., verifying the existence of an edge) and can hinder the computation of list intersections. Analyzing existing systems and queries from prominent graph benchmarks [20, 31, 52], we have identified five representative low-level queries, detailed in Table 2. These queries include: (1) Edge Existence Check: determines whether there is an edge that connects two specified vertices. (2) 1-Hop Neighborhood Scan: provides a list of all vertices directly neighboring a given vertex. (3) 2-Hop Neighborhood Scan: computes the union of the neighbor lists of all neighbors of a given vertex. (4) Cycle Detection: verifies the presence of at least one cycle between two vertices. (5) Clique Search: identifies all cliques composed of unique vertices that include a specified vertex. In actual systems, there is usually a parameter k to constrain the size of cliques.

Table 2: Key query primitives for DGM

Primitive	Complexity	Category	Input
$insertE(u,v)$	$O(log(d(v)))$	update	edge
$insertV(v)$	$O(log(\|V\|))$	update	vertex
$deleteE(u,v)$	$O(log(d(u)))$	update	edge
$deleteV(v)$	$O(d(v) * log(d(v))$	update	vertex
$get1hop(v)$	$O(d(v))$	query	vertex
$get2hop(v)$	$O(d(v) * \bar{d})$	query	vertex
$findClique(v)$	NP-hard	query	vertex
$findCycle(u,v)$	$O(\|V\|)$	query	vertex pair
$findEdge(u,v)$	$O(log(\|d(u)\|))$	query	vertex pair

3.2 Design Choices and Optimization

In Section 3.1, we explore the commonalities among existing GPU-based DGM systems. Therefore, in this subsection, we will discuss what makes specific GPU-based DGM systems stand out. We summarize design choices and optimization directions into three perspectives: data structures, memory management, and tailored optimization.

3.2.1 Data structures. Choosing effective structures for topology and attribute data is a primary goal for DGM developers. An ideal data structure incorporates an optimal data layout and implementation of primitives to facilitate quick updates and efficient querying within acceptable memory usage. However, trade-offs are inevitable among these factors. For instance, using a dense matrix format for graph data allows for constant-time edge insertion but results in more memory consumption. Alternatively, adopting sorted arrays for neighbor lists increases its insertion overhead due to the need to maintain sequential storage. Finding a one-size-fits-all solution appears challenging. Instead, the optimal choice varies with different workloads.

3.2.2 Memory management. Optimizing memory allocation and management is crucial due to the evolving and unpredictable volume of dynamic graphs in real-world scenarios. When a system exhausts its pre-allocated space, it must expand its data storage, and similarly, contract when excess space is available. However, the frequent allocation and freeing of memory necessitate time-consuming synchronization between the CPU host and GPU kernels. Moreover, relocating existing data incurs additional costs. Therefore, developing appropriate mechanisms for memory resource management has been another key focus of prior GPU-based DGM systems.

3.2.3 Tailored optimization. Detailed optimization of bottleneck operations, such as edge updates and list intersections, requires analyzing both algorithmic behaviors and hardware capabilities. Certain systems employ auxiliary indices to quicken essential operations, such as determining an insertion point [9]; others implement advanced concurrency control strategies to minimize read/write conflicts [71]; while others take advantage of GPU-specific features such as shared memory to buffer frequently accessed data [4].

4 COMPARISON OF COMPETING SYSTEMS

We evaluated six GPU-based DGM systems: cuSTINGER [29], Hornet [9], Gunrock [62], faimgraph [65], GPMA [56], and LPMA [71]. Among existing systems, they achieve state-of-the-art performance and succeed in handling workloads in our evaluation. In the following section, we introduce data structures, memory management, and tailored optimization of each system based on our model in Section 3 and Table 3 summarizes key design decisions mentioned here.

4.1 cuSTINGER

Data structure: cuSTINGER [29] represents an early exploration of GPU-based DGM. Specifically, it is an improved version of STINGER [21] optimized for GPU acceleration using CUDA. Notably, cuSTINGER adapted the data structure of STINGER to leverage the parallel processing capability of GPUs. Illustrated in Fig. 2 (a), cuSTINGER

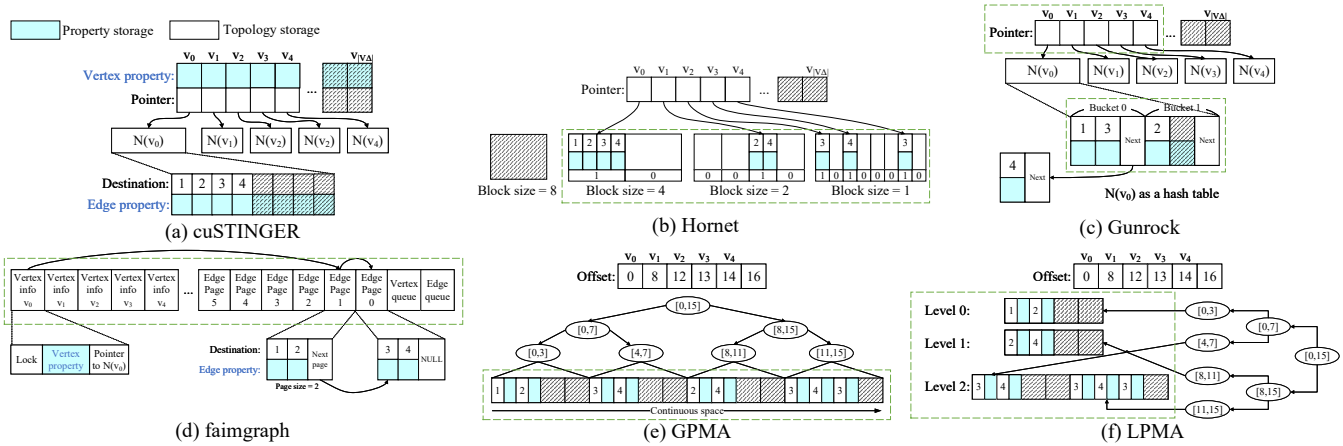


Figure 2: Overview of compared systems storing an example graph, where $N(v_0) = \{1, 2, 3, 4\}, N(v_1) = \{2, 4\}, N(v_2) = \{3\}, N(v_3) = \{4\}, N(v_4) = \{3\}$. Pre-allocated parts are in dotted boxes. Preserved space for future insertion is covered by black slashes.

utilizes a semi-dense array to store a neighbor list associated with a specific vertex, which means that cuSTINGER allocates extra space within its array to accommodate future data. Structured as a Structure of Arrays (SOA), this data organization allows for compaction akin to CSR format, leading to decreased memory demands and enhanced data locality.

Memory management: Upon reaching the full capacity of a neighbor list array, the system dynamically allocates additional space and subsequently transfers existing data to the expanded area. Users are granted the flexibility in cuSTINGER to specify the allocated memory size or enable dynamic memory allocation based on their requirements. To cater to varying application needs, cuSTINGER can adjust its meta-data modes to optimize GPU space usage based on specific graph properties.

Tailored optimization: cuSTINGER focuses on improving edge insertion and deletion. It separates deletion and insertion for the convenience of parallelization. Receiving a batch of insertion, the system first conducts a parallel search to eliminate the duplicate edges and repeated insertion.

4.2 Hornet

Data Structure: Hornet [9] utilizes a series of fixed-size block arrays to store neighbor lists. This data layout is demonstrated in Fig. 2 (b). The block sizes are defined as powers of two (e.g., 1, 2, 4, 8, etc.). Hornet ensures that each neighbor list is stored in the smallest block that accommodates it. For instance, upon inserting a new edge to a neighbor list of 2 elements, the revised list will be transferred to a block array with a block size of 4. Hornet maintains two auxiliary structures to boost insertion. First, Hornet speeds up the discovery and recovery of empty blocks using a vectorized bit tree. Second, Hornet maintains a collection of B+ trees to check the availability of block arrays faster. Each B+ tree is dedicated to managing a block array of a specific block size.

Memory Management: At the initialization stage, Hornet reserves substantial memory for block arrays, which is managed by its memory pool. When an updated list overflows, a new block is assigned and Hornet must copy old data to this block. If no available block exists, a new block array will be allocated.

Tailored optimization: Hornet operates under the assumption of non-overlapping updates and queries, managing them sequentially. For updates, it employs a bulk synchronization approach to optimize parallelism.

Table 3: Comparison of design decisions

systems	Basic structure	Pre-alloc[a]	Re-org[b]	Ext[c]
cuSTINGER [29]	adj	×	×	✓
faimGraph [65]	adj	✓	×	×
Hornet [9]	adj	✓	✓	✓
Gunrock [62]	hash + adj	×	×	✓
GPMA [56]	CSR + PMA	×	✓	✓
LPMA [71]	CSR + PMA	×	✓	✓

[a] pre-allocate a large amount of memory
[b] move data on-the-fly for reorganization
[c] extension when overflow

4.3 Gunrock

Data structure: Originally, Gunrock [62] works as a GPU-based graph processing framework designed for static graphs. The developers integrated their GPU-based hash table [4] into Gunrock by modifying the graph storage to accommodate dynamic graph workloads, as shown in Fig. 2 (c). A vertex dictionary maintains pointers to neighbor lists stored as hash tables. The hash table, called Slab Hash [4], employs a fixed number of N buckets arranged in consecutive memory locations within a direct-address table.

Memory management: Initially, hash buckets do not have any successors. When a hash collision occurs and a bucket lacks sufficient space to accommodate additional elements, the system automatically allocates a new bucket. It then updates the successor pointer of the full bucket to direct to the newly allocated bucket, thereby establishing the first N buckets as the head nodes of N interconnected linked lists. The process of bucket allocation is managed by a memory manager, which governs a hierarchical memory structure comprising multiple memory pools.

Tailored optimization: Gunrock prioritizes the optimization of edge insertion and deletion primitives. Its implementation relies

on warp-level primitives for efficient data communication and synchronization. Gunrock operates under the assumption that each thread handles a single edge insertion task. Edges sharing the same source vertex are grouped for processing in a coalesced replace operation on the relevant hash table. Deleted edges are marked as invalid without immediate removal. Marked edge locations are treated as non-empty during edge insertion and are subsequently cleared from the data structure.

4.4 faimgraph

Data structure: Fig. 2 (d) gives a demonstration of faimgraph [65], which is an improved version of aimgraph[66]. Regarding vertex data, they are stored in a dynamic growing array within faimgraph. Every vertex contains a collection of workload-related parameters and a reference to its corresponding neighbor list. As for edge data, faimgraph's data storage architecture resembles that of adjacency lists. Each neighbor list resides on one or more designated pages. Pages are of a fixed size, and multiple pages form a linked list to accommodate neighbor lists larger than a single page. Besides, faimgraph features auxiliary structures to expedite memory reclamation. Specifically, faimgraph utilizes two indices to cache freed vertex blocks and edge pages, respectively. When seeking available space, a thread attempts to obtain a portion of free space from the queue-like indices before resorting to initiating a direct allocation request.

Memory management: The primary focus of faimGraph is reducing the overhead associated with memory allocation and deallocation on GPUs, as these processes often disrupt the execution of GPU kernels due to requiring CPU intervention for synchronization. During system setup, faimGraph allocates a substantial amount of memory and structures it using a tailored memory management approach. Thus, all memory operations occur on the GPU side, eliminating the need for reallocation or re-initialization once faimGraph is running.

Tailored optimization: For vertex insertion and deletion, faimgraph employs a method called reversed duplicate check. This involves assigning each thread a vertex ID and tasking it with performing a binary search among the vertices scheduled for insertion. Due to the significantly smaller number of vertices scheduled for insertion compared to the total graph vertices, conducting a binary search among them is efficient, and the increase in search operations can be offset by leveraging extensive parallel processing capabilities. For edge insertion and deletion, faimgraph pre-processes an update batch to group operations based on source vertices and eliminate duplicates within the batch.

4.5 GPMA

Data structure: As shown in Fig. 2 (e) GPMA [56] utilizes the Packed Memory Array (PMA) [8] to manage neighbor lists within CSR representations for dynamic graphs. PMA maintains an array of fixed-size segments while incorporating gaps dynamically organized by a self-balancing tree to facilitate rapid updates without violating a predefined threshold of load factor in each segment. GPMA extends the functionality of PMA to the GPU platform, employing it to store arrays in a self-balancing CSR-like structure. As mentioned before, LPMA adopts a tiered tree-like structure. Developers of LPMA align their design with the GPU's memory hierarchy, using shared memory to store the first few levels of the self-balancing tree, because the first few levels are accessed by almost every update, thus being the most frequently accessed data.

Memory management: As the number of inserted edges increases, causing GPMA's load factor to exceed a predefined threshold, an expansion process is initiated. During expansion, a request for additional memory space is made, and the complete data structure is relocated to this new memory location. Expansion becomes inefficient when dealing with high-volume update operations in dynamic graphs.

Tailored optimization: GPMA employs a bottom-up update strategy for updates, starting with finding a leaf layer segment to be inserted. During each segment update, a thread or a warp (depending on the segment size) is assigned to manage the update process. GPMA initiates an expansion process that doubles the size of the original sorted array. Subsequently, GPMA combines all edges from the original array with the inserted edges and redistributes them into segments in the expanded array.

4.6 LPMA

Data structure: LPMA [71] represents an enhanced version of GPMA, illustrated in Fig. 2 (f), where the block array containing edge data is partitioned into multiple levels. Successive levels increase in size exponentially based on a specified integer exponent (typically 2) within LPMA.

Memory management: Expansion in LPMA involves allocating a new level and appending it to the existing LPMA tree instead of reallocating a doubled continuous space as done in GPMA. This method reduces memory allocation costs and mitigates memory fragmentation problems encountered in GPMA. Following expansion, LPMA implements a top-down re-balance strategy that notably reduces unnecessary re-balancing operations. In contrast to GPMA's global re-balancing approach involving all edges across the entire expanded space, LPMA focuses on re-balancing edges within necessary segments.

Tailored optimization: LPMA processes queries and updates in a batch sorted by their corresponding timestamps. It separates the batch into an update group and a query group and deals with the update group before the query group. LPMA stores edges with the same source and destination but different timestamps during batch execution to ensure consistency and merge them after finishing a query batch.

5 EXPERIMENT

5.1 Experiment Setup

Environment. All experiments are conducted on a Linux server with two Intel(R) Xeon(R) Gold 5218 CPUs and enough main memory (1T) to store the entire graph data. GPU-based systems are tested using Nvidia A100s with 40GB of GPU memory. We use the CUDA 11.3 toolkit to compile the GPU projects.

Compared Systems. We evaluate six state-of-the-art DGM solutions on the GPU: `cuSTINGER`[29], `Hornet`[9], `faimGraph`[65], `Gunrock`[62], `GPMA`[56], and `LPMA`[71]. We collect codes from their authors for performance evaluation. We also include a CPU-based method[25] that achieves the best performance in our evaluation

against other CPU-based systems to provide an absolute baseline. Some DGM systems (i.e., cuSTINGER) do not provide inherent support for the evaluated analytic workloads, so we implement these algorithms on top of them by ourselves.

Settings. In all evaluations, once inserted, graph data including topology data and attributes are entirely stored on GPU memory. Operation batches including updates and queries are stored in CPU memory at first and then transferred to GPU kernels. An edge update consists of two vertex IDs, consuming 8 Bytes (two 32-bit unsigned integers). The time for cross-device data transfer of data graphs and operation batches is excluded in the reported latency.

Table 4: Details of Datasets

dataset	$\|V\|$	$\|E\|$	AD[a]	source
road	1.3M	1.9M	1.39	road network
Wiki	1.1M	3.3M	2.9	web
Patent	3.7M	17M	4.38	citation
Pokec	1.6M	30M	18.75	social
LiveJournal	4.8M	68M	14.3	social
Stack	2.6M	36M	13.93	web
Graph500	1M	125M	124.41	synthetic
Orkut	3M	117M	38.14	social
LDBC-SF30	106M	701M	6.61	synthetic
uk2005	39M	936M	23.79	web
LDBC-SF100	337M	2.2B	6.52	synthetic

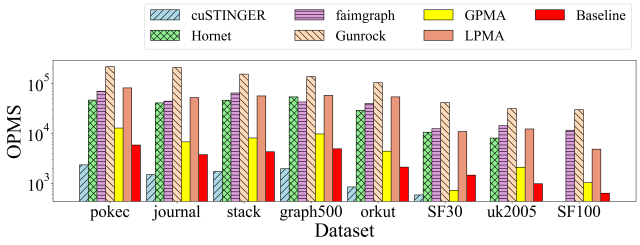
[a] Average degree

Datasets. We choose datasets that are commonly used in previous work, obtained from SNAP[1], Graph500 benchmarks[2] and LDBC SNB interactive benchmark[52]. They cover both real-world and synthetic datasets. The number of edges varies from millions to billions to evaluate the scalability of the compared system. The details of our datasets are given in Table 4. We use real-world graphs from different sources, including road networks, web pages, citation networks, and social media. One of the synthetic datasets we use is a power-law graph generated by RMAT from Graph500 benchmarks. We also include a data generator used in the LDBC SNB interactive benchmark, which simulates a real-world social network. The tested datasets are representative, covering highly skewed, uniform-distributed, and massive graphs. For datasets without timestamps attached, we assign a random timestamp to each edge, as done in most existing work.

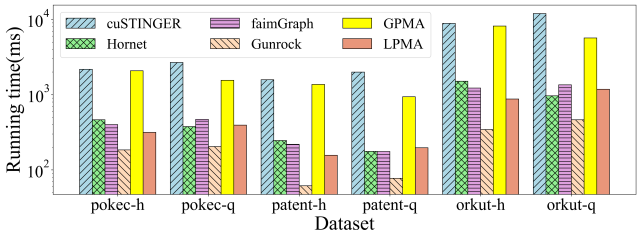
5.2 Update performance

In this section, we vary batch sizes from 10^4, 10^5, to 10^6, adhering to the prevalent configurations in existing work, and evaluate systems' throughput, scalability, and stability. When the batch size is smaller than 10^4, gaps between these systems are too small. When the batch size is larger than 10^6, it may cause failure for some systems (`cuSTINGER` and `Hornet`). Besides, we include two batch settings different from any previous studies. First, we construct dynamic workloads with batches of different sizes using a time-window-based method. Second, we test existing systems' performance to handle update-query mixed workloads with various update/query ratios.

[1]http://snap.stanford.edu/data/
[2]https://graph500.org/



(a) Peak throughput with fixed batch sizes of 10^4, 10^5, to 10^6



(b) Latency with dynamic batch sizes. Suffix "-h" means hot periods and "-q" refers to quiet periods.

Figure 3: Update performance of different batch size settings.

5.2.1 Update-only workloads. We test all GPU-based systems and the CPU-based baseline with three batch sizes (10^4, 10^5, to 10^6) and calculate peak throughput in Operation Per Millisecond (OPMS), as reported in Fig.3(a). Specifically, `Gunrock` outperforms the competitors on all datasets, achieving an advantage from 3.55× to 122.6× against other systems, whose advantage is predictable because it is the first to leverage hash tables to store neighbor lists. In `Gunrock`'s implementation, each vertex has an independent hash table to keep its neighbors, which is optimized for GPU processing and features independent updates of different hash buckets. `faimgraph`, `LPMA` and `Hornet` achieve similar throughput but only `Hornet` achieve peak performance at smaller batch size. `GPMA` and `cuSTINGER` have trouble with large graphs, throwing errors, or showing performance loss. Besides, on those more skewed graphs, `cuSTINGER` are sometimes beaten by the CPU baseline, while other systems prevail the baseline. Unlike `Gunrock`, other systems split the neighbor lists into pages or block arrays with reserved space for data to be inserted. This brings two trade-offs in DGM performance tuning. First, with more new edges, the reserved space may run out or become imbalanced. Then, expansion or re-organization of data structures becomes necessary but they will introduce overhead coming from data movement, memory allocation, and synchronization. This is the reason why `Hornet` and `cuSTINGER` fall behind when graph size grows and thus expansion is more frequent. However, we can not relentlessly allocate more space at first, which may waste precious GPU memory. Second, keeping sorted neighbor lists brings convenience to list intersections, as shown in Fig. 8. However, it may turn out to be penalties for updates due to more data movement and complicated conflict control.

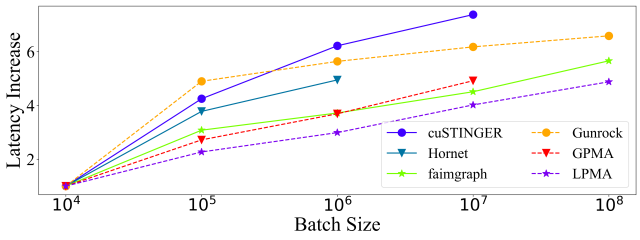


Figure 4: Normalized latency increases as the batch size grows.

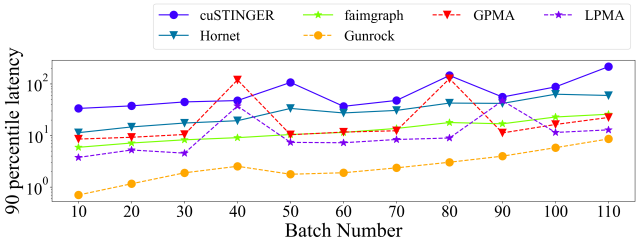


Figure 5: Stability of update latency on Orkut.

In Fig. 3(b), we show the latency for processing all dynamic-size batches. To generate workloads with various batch sizes, we use a time window to partition a stream of update operations with their timestamps. Specifically, suppose the time range of a dataset is $[t_{min}, t_{max}]$, where t_{min} and t_{max} are the smallest and largest timestamps, respectively. We divide the time range into g time windows with equal time intervals so that updates that fall into the same time window form a batch. We control the value of g and timestamp distribution to generate hot periods with mostly large batches and a quiet period with more small batches. Experimental results show that Gunrock still prevails due to its suitability for parallel updates. Hornet and cuSTINGER have trouble dealing with hot periods while LPMA, GPMA, and faimgraph is faster in hot periods than quiet periods. These observations confirm our statements in the last paragraph.

5.2.2 Scalability over batch size. Apart from comparing the latency, we are also interested in scalability over batch size. An essential advantage of GPU-based DGM systems is parallelism, and a larger batch should bring more chances to leverage more threads and achieve higher memory bandwidth utilization.

In Fig. 4, we choose a large dataset and show how latency changes as batch size increases. When the batch size increases by 10×, LPMA shows the smallest latency growth thanks to its redundant-free re-balance and update strategies. The scalability of Hornet and GPMA is worse than other systems. When a certain amount of edges has been inserted, Hornet has to find a new block to store the enlarged neighbor list, and then move the data to the new place. GPMA needs to invoke a re-balance procedure to rearrange data to ensure that reserved space in every block is in a pre-defined threshold. Memory allocation and data movement inside the data structure will seriously stall active threads. As for Gunrock, concurrent access to hash tables leads to more stalled random IO requests, so it suffers from more performance loss as the batch size grows from 10^4 to 10^5, but the growth of its latency plateaus because of achieving higher parallelism.

5.2.3 Stability. Apart from update speed and scalability, desired DGM systems should strive to minimize substantial variations in latency across various batches, especially as long-term data management services. Usually, the significant fluctuation in latency may bring trouble to the downstream applications, ultimately compromising the entire system's performance. We report the 90 percentile latency of every 10 consecutive batches, as shown in Fig. 5. From the latency curves, we have three observations.

First, among the compared systems, Gunrock and faimgraph have the best stability, whose standard deviations of 90 percentile latency are low in Fig. 5. This is because faimgraph allocates a large amount of memory beforehand and designs a memory management technique, which entirely runs on the GPU side to avoid intervention or reallocation procedures from the CPU host. faimgraph organizes the neighbor lists as fixed-size pages, so allocating a new page during insertion is cheap. Gunrock uses hash tables based on an array of hash buckets and linked lists for collision, so insertions are mapped to random locations, and extending a new node in a linked list is less costly. Second, the stability of compared systems deteriorates to different extents as the graph data scale grows. For example, when more data have been stored, Hornet has a higher chance to invoke data movement, migrating the neighbor list to bigger memory blocks. Third, we can see obvious crests on the latency curves of cuSTINGER, GPMA, and LPMA. cuSTINGER uses a brute strategy of over-allocating the neighbor lists. If a neighbor list is full, cuSTINGER re-allocates double-sized memory space and moves the original data to the new space. GPMA and LPMA apply a tree-based structure to balance the reserved empty slots, but its expansion becomes a bottleneck. As an improved version of GPMA, LPMA finishes the re-organization faster because it is a layered data structure, so fewer data are influenced and moved at each expansion.

5.2.4 Impact of skewness. In update-only evaluations, we notice that system performance varies in data graphs of the same scale but with different levels of skewness. Therefore, we investigate the impact of skewness in this subsection. We generate a set of data graphs with the same number of edges but various skewness by re-distributing edges in a highly skewed data graph, and report update latency normalized by original data in Fig. 6. With the diminution of skewness, latency is observed to decrease across all comparing systems because large neighbor lists lead to more frequent expansion (cuSTINGER), worse workload balance (faimgraph), more re-organization (Hornet, LPMA and GPMA), and more hash conflicts (Gunrock). Among these factors, frequent expansion leads to the most severe performance loss. Besides, this trend is also clear in tree-like data structures, such as LPMA and GPMA. Large neighbor list sizes (i.e., higher skewness) result in more insertion to some particular regions of the entire pre-allocated space, exceeding the density threshold and triggering expensive local re-organization.

5.2.5 Mixed workloads. Fig. 7 demonstrates system performance on update-query mixed workloads. To generate such workloads,

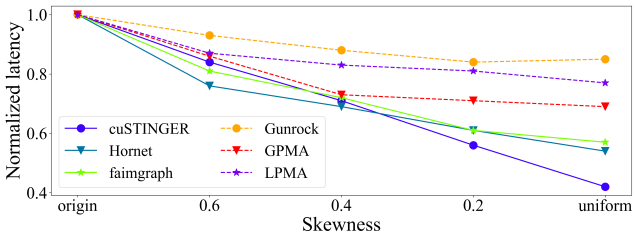


Figure 6: Change of normalized update latency for different levels of skewness

we first choose three update-query ratios: 25%, 50%, and 75% updates. The selection of query/update ratios is based on an analysis of previous work [53] from the industry and well-known graph benchmarks such as LDBC SNB interactive workloads[52]. We use edge existence check and neighborhood scan as possible queries here. Apart from LPMA, other existing systems do not provide native support for mixed workloads, suffering from the overhead of switching between update and query logic. Therefore, undoubtedly LPMA is the champion in mixed workload evaluation, which reorders a mixed batch and conducts updates before queries to avoid contention. However, reordering is a straightforward optimization and can be easily adopted in other systems but we can not see more sophisticated optimizations in existing systems. Thus, studies in mixed workload support on GPUs are inadequate and can be a promising future topic.

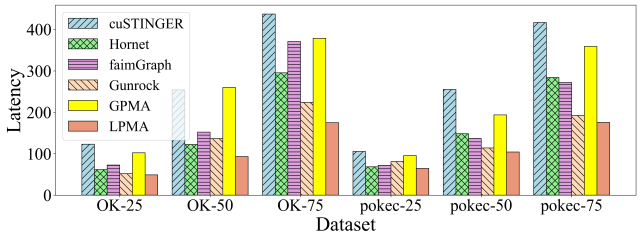
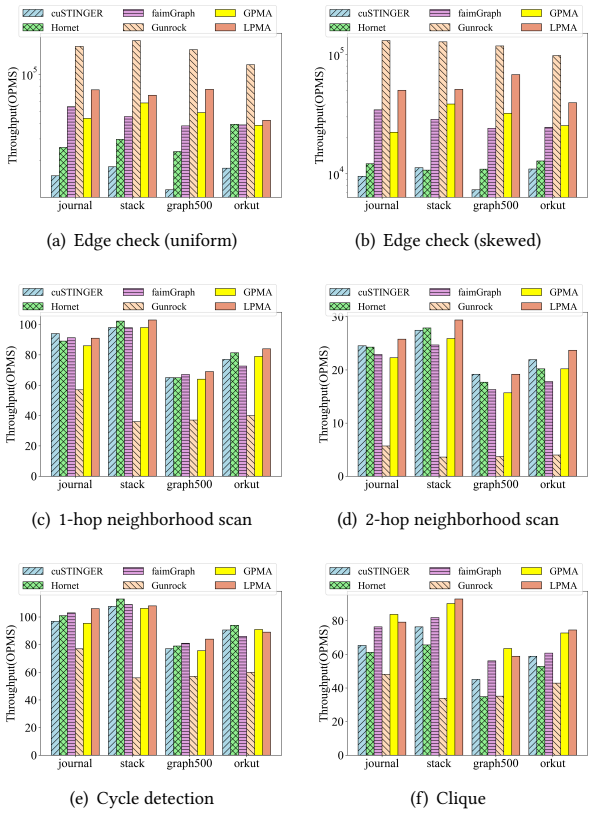


Figure 7: Performance on mixed workload with various update-query ratios.

5.3 Query performance

We generate workloads for different primitives in the following ways. For edge existence checks, we sample vertex pairs with equal probability to generate a uniform workload. Then, we sample vertex with $P(v) = \frac{|d(v)|}{2|E|}$ to generate a workload influenced by the skewness of data graphs, where vertices with higher degrees are more likely to be accessed. We uniformly sample vertices for 1-hop neighborhood scans, 2-hop neighbor scans, and clique searches and require the system to return 1-hop or 2-hop neighbor lists or clique collections containing a given vertex. For cycle detection, we use uniformly distributed vertex pairs in the edge existence check test and ask systems whether they are in at least one cycle. We limit the size of cliques and cycles to 4 by default.

5.3.1 Edge existence check: Fig. 8(a) and 8(b) compares the throughput of the edge existence check. Gunrock is far ahead on a uniform



(a) Edge check (uniform) (b) Edge check (skewed)

(c) 1-hop neighborhood scan (d) 2-hop neighborhood scan

(e) Cycle detection (f) Clique

Figure 8: Performance of different query primitives.

workload, while on a skewed workload, its lead is even bigger because large neighbor lists bring more cost to scan or conduct binary searches on neighbor lists than hash tables, not to mention more queries accessing large neighbor lists. Hence, the advantage of hash tables against arrays in edge existence check is obvious. Besides, among array-based solutions, those maintaining sorted neighbor lists for fast searching achieve higher throughput than others.

5.3.2 1-hop and 2-hop neighborhood scan: Fig. 8(c) gives results of 1-hop neighborhood scans, which is quite the opposite of our edge existence checks. Gunrock performs the worst on neighborhood scans and cuSTINGER, which is not competitive for update latency and edge existence check, achieves the best throughput on some datasets. Other systems are close in terms of 1-hop neighborhood scan performance. Results in Fig. 8(d) involving 2-hop neighborhood scan show a similar situation where performance gaps between Gunrock and other systems are even larger. Such performance loss of Gunrock results from its hash-based data storage, which requires extra random memory access to jump via pointers and overhead to skip empty slots when scanning a neighbor list.

5.3.3 Clique search and Cycle detection: Fig. 8(f) and 8(e) demonstrate the results of clique search and cycle detection tests. We discuss these two queries together because they have similar behavior of depth-first-search in our implementation. A clique search also runs neighbor list intersections to filter out neighboring vertices unlikely to be in a clique. Although the overall performance

ranking is the same as one in neighborhood scant tests, Gunrock closes the gaps in both queries because, for cycle detection, we can stop searching as long as we find one cycle, and, for clique search, list intersections using pre-constructed hash tables can achieve high parallelism. Another interesting observation is that systems keeping sorted lists perform better on clique search because they facilitate faster list intersections.

5.4 Performance of Analytic Workloads

We select three representative algorithms (BFS, Pagerank, and Betweenness Centrality) to compare the analytical performance of existing systems. Some tested systems, by default, initially transform the data structure into a static layout for static graphs (typically CSR). They then essentially execute the workload based on a static layout. Hence, we incorporate the time for transforming data representations into latency as penalties. However, in circumstances where updates are less often, conversion to static structures is not a bad idea.

5.4.1 BFS. BFS tests the systems' capability to scan the neighborhood of vertices, which is selected by most existing GPU-based DGM solutions as an analytical workload in their experiments. Fig. 9(a) presents the latency for BFS. The results correspond with the evaluation of neighborhood scans to some extent. Generally, GPMA, LPMA and cuSTINGER have the best performance on different datasets, while the gaps among other systems are small. Suffering from penalties of CSR conversion, Gunrock can be slower than other systems, especially for larger graphs.

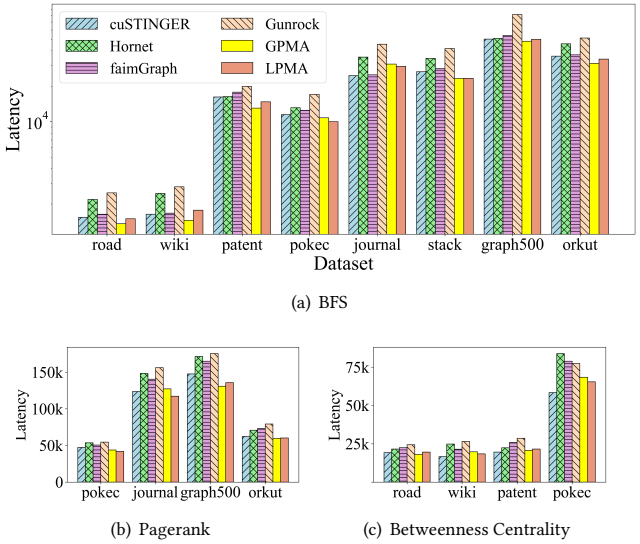


Figure 9: Performance of BFS, Pagerank, and Betweenness Centrality

5.4.2 Pagerank. Pagerank also launches neighborhood scans, but unlike BFS, Pagerank accesses more data because it reads the neighborhood of every vertex at every iteration. What's more, the order of accessing the neighborhood can be fixed, usually following the order of the vertex IDs (i.e. from v_0 to $v_{|V|}$). Fig. 9(b) demonstrates the latency comparison for Pagerank. GPMA, LPMA, and cuSTINGER are still the best three systems with very close performance. Their advantage against other systems is larger than that in the BFS test because they store the neighbor lists in the same order as the vertex IDs, which improves their memory access locality. Gunrock is again of the worst performance due to its trouble in neighborhood scan.

5.4.3 Betweenness Centrality. Betweenness Centrality has different behaviors from BFS and Pagerank. The major computational cost of Betweenness Centrality comes from calculating amounts of the shortest path passing a specific vertex, which can be seen as conducting BFS with a vertex as a source. The algorithm compares distances between two visited vertices and the source vertex so it requires access to edge attributes. As shown in Fig.9(c), Gunrock narrows its gap because of its efficiency for acquiring edge attributes.

5.5 Hardware utilization

5.5.1 Memory consumption. Given that memory resources on GPUs are more limited and expensive compared to CPU memory, a DGM system compromising memory efficiency in favor of throughput too much would not constitute a desirable design. As shown in Fig 10, all systems lead to different levels of memory increase compared with compact CSR. cuSTINGER usually has a small memory footprint because it only expands a neighbor list when it is almost full. However, its memory efficiency is severely influenced by expansion. If the last few insertions to a list trigger an expansion, its memory footprint will grow significantly. Gunrock requires extra pointers to maintain hash tables. Hornet and faimgraph use aggressive memory management strategies, allocating a large portion of memory in advance and maintaining extra pointers or auxiliary structures. Hornet guarantees that a neighbor list is kept in one fixed-size block, whose size is extended by a power of 2 when a neighbor list violates its size limit. Therefore, it consumes more memory especially when large neighbor lists exist.

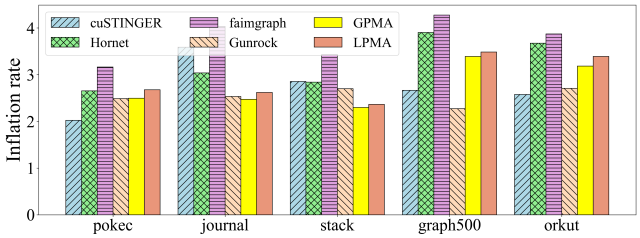


Figure 10: Memory inflation rate against CSR

5.5.2 Roofline analysis. The roofline model [64] is famous for its effectiveness in investigating bottlenecks in hardware resources like compute units, DRAM, and Caches. No previous studies have conducted such analysis to figure out possible optimization of resource utilization. We use Nvidia's official profiling toolchains to obtain kernel execution metrics including integer operations count, total bytes from HBM, and execution duration to build a roofline model. As shown in Fig. 11, existing systems are memory-bound and can not make full use of memory bandwidth for two reasons. First, some systems involve hash addressing, backtracking in a tree

structure, or jumping via pointers, all of which lead to random accesses. Second, incoming insertions may cause memory allocation and block computation logic.

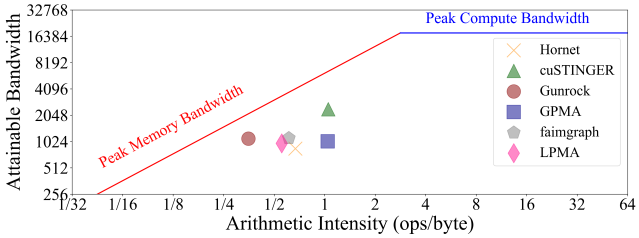


Figure 11: Roofline analysis on batch updates

5.6 Extensions

5.6.1 Multi-GPU support. Modern computation platforms often have more than one GPU, but existing systems do not pay attention to DGM on multiple GPUs, which is a research gap worthy of investigation. Hence, we present experiments in Fig. 12 by incorporating existing systems and graph partitioning methods[28, 40, 42]. If an application requires mostly updates and simple queries like edge existence checks, a naive partition-based method works well, as in Fig. 12 read-only workload and edge existence checks suffer from small performance loss. However, matching complicated patterns like clique and 2-hop neighbors turns out to be costly and can even be slower than single-GPU implementation. Therefore, designing multi-GPU DGM systems requires careful design when complex computation is needed, where the key problem is dealing with cross-partition edges because they are likely to be accessed in complicated pattern matching but not in edge existence checks.

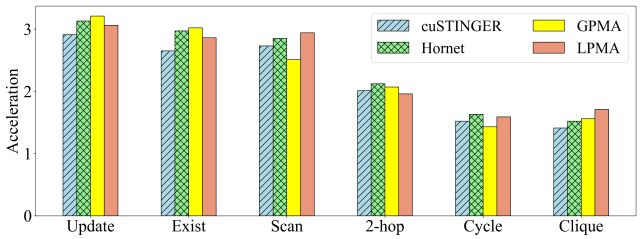


Figure 12: Acceleration ratios of primitives of single-GPU against 4-GPU implementations

5.6.2 Concurrency support. As mentioned in Section 5.2.5, only one existing system LPMA provides native support for a mixed workload by separating updates and queries to avoid contention. This is a tricky optimization but we are interested in how GPU-based DGM systems can achieve concurrency support with overlapping queries and updates, which is a common topic in studies on concurrent data structures and databases. That is the reason why we carry out the experiment in Fig. 13, where we modify faimgraph and LPMA, implementing lock mechanisms of different granularity using atomic operations in CUDA. We add a coarse-grained lock to the entire balanced tree in LPMA but fine-grained locks to each data page in faimgraph. Relative runtime of concurrent LPMA increases significantly but faimgraph achieves better mixed workload performance than its original version. Hence, data structures that are friendly to fine-grained concurrency control can be promising.

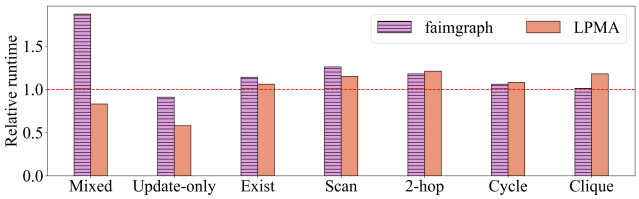


Figure 13: Performance change of LPMA and faimgraph with lock mechanisms for concurrency control

6 CONCLUSIONS

In this paper, we study the performance of six GPU-based DGM systems, analyzing the motivation and advantages of each design. We also conduct experiments on both real-world and synthesis graph datasets, scaling from millions to thousands of millions of edges. We use a unified model, metrics, and workloads to evaluate update latency, resource utilization, and memory consumption in a fair condition. Therefore, we can have a more comprehensive view when comparing existing systems.

Recommendations and future direction: We have the following recommendations on design choices or unsolved problems for future studies. (1) If the workload is update-heavy, hash tables will be a good alternative to array-like structures to store the neighbor lists. Users can lessen the disadvantage of neighborhood scan by adopting a more compact hash table design (i.e., higher load factor, fewer pointers). (2) The most important factor for choosing a proper DGM system is the behavior of workloads. For example, keeping sorted arrays can benefit list intersection but slow down updates. Since there is no one-size-fits-all design, using hybrid data structures can be a way out. (3) If real-time responses are not strictly required, or the overlapping of updates and computation is small, transferring the DGM structure to a compact data layout will be a good idea. (4) Be careful when you decide to rearrange the data during execution, which results in unstable performance. Based on existing studies, allocating a large space and designing customized memory management is helpful. (5) Existing studies focus on exploiting peak performance, and thus some settings like batch updates without overlapping with queries may not be practical. Concurrency control and efficient query-update mixed workload are worth considering in future work to implement truly useful DGM systems on GPUs. (6) Hardware-related metrics like memory hierarchy and resource utilization are not well investigated in current systems. Insights from related communities like database design should provide valuable inspiration for us to develop a better GPU-based DGM system.

ACKNOWLEDGMENTS

This work was partially supported by the National Key Research and Development Program of China under grant 2023YFB4502303. Lei Zou is the corresponding author.

REFERENCES

[1] Tariq Abughofa and Farhana H. Zulkernine. 2018. Sprouter: Dynamic Graph Processing over Data Streams at Scale. In *International Conference on Database and Expert Systems Applications*.

[2] Hisham Alasmary, Aminollah Khormali, Afsah Anwar, Jeman Park, Jinchun Choi, Ahmed Abusnaina, Amro Awad, DaeHun Nyang, and Aziz Mohaisen. 2019. Analyzing and Detecting Emerging Internet of Things Malware: A Graph-Based Approach. *IEEE Internet Things J.* 6, 5 (2019), 8977–8988. https://doi.org/10.1109/JIOT.2019.2925929

[3] Khaled Ammar. 2016. Techniques and Systems for Large Dynamic Graphs. In *SIGMOD PhD Symposium*. ACM, 7–11.

[4] Saman Ashkiani, Martín Farach-Colton, and John Douglas Owens. 2017. A Dynamic Hash Table for the GPU. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2017), 419–429. https://api.semanticscholar.org/CorpusID:3676148

[5] Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *2018 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 419–429.

[6] Muhammad A Awad, Saman Ashkiani, Serban D Porumbescu, Martín Farach-Colton, and John D Owens. 2023. Analyzing and implementing GPU hash tables. In *2023 Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 33–50.

[7] Scott Beamer, Krste Asanović, and David A. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. *2015 IEEE International Symposium on Workload Characterization* (2015), 56–65.

[8] Michael A. Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Trans. Database Syst.* 32, 4 (nov 2007), 26–es.

[9] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *HPEC*. IEEE, 1–7.

[10] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating real-time graph mining. In *Cloud Data Management*.

[11] Huanqi Cao, Yuanwei Wang, Haojie Wang, Heng Lin, Zixuan Ma, Wanwang Yin, and Wenguang Chen. 2022. Scaling graph traversal to 281 trillion edges with 40 million cores. In *PPoPP*. ACM, 234–245.

[12] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (2023), 441–454.

[13] Shuai Che. 2014. GasCL: A vertex-centric graph model for GPUs. In *HPEC*. IEEE, 1–6.

[14] Shuai Che, Bradford M. Beckmann, and Steven K. Reinhardt. 2014. BelRed: Constructing GPGPU graph applications with software building blocks. In *HPEC*. IEEE, 1–6.

[15] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *OSDI*. USENIX Association, 857–877.

[16] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *European Conference on Computer Systems*. https://api.semanticscholar.org/CorpusID:3721244

[17] Philip Dexter, Yu David Liu, and Kenneth Chiu. 2016. Lazy graph processing in Haskell. *Proceedings of the 9th International Symposium on Haskell* (2016).

[18] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *PLDI*. ACM, 918–934.

[19] Mengsu Ding, Muqiao Yang, and Shimin Chen. 2019. Storing and Querying Large-Scale Spatio-Temporal Graphs with High-Throughput Edge Insertions. *ArXiv* abs/1904.09610 (2019).

[20] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and O. Udrea. 2011. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *ACM SIGMOD Conference*. https://api.semanticscholar.org/CorpusID:14768738

[21] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. *2012 IEEE Conference on High Performance Extreme Computing* (2012), 1–5.

[22] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*. ACM, 8–21. https://doi.org/10.1145/2274576.2274578

[23] Guanyu Feng, Zixuan Ma, Daixuan Li, Xiaowei Zhu, Yanzheng Cai, Wentao Han, and Wenguang Chen. 2020. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. *Proceedings of the 2021 International Conference on Management of Data* (2020). https://api.semanticscholar.org/CorpusID:214775005

[24] Guoyao Feng, Xiaomin Meng, and Khaled Ammar. 2015. DISTINGER: A distributed graph data structure for massive dynamic graph processing. *2015 IEEE International Conference on Big Data (Big Data)* (2015), 1814–1822.

[25] Per Fuchs, Jana Giceva, and Domagoj Margan. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15 (2022), 1173–1186.

[26] Hongru Gao, Xiaofei Liao, Zhiyuan Shao, Kexin Li, Jiajie Chen, and Hai Jin. 2024. A survey on dynamic graph processing on GPUs: concepts, terminologies and systems. *Frontiers Comput. Sci.* 18, 4 (2024).

[27] Joseph E. Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX Symposium on Operating Systems Design and Implementation*.

[28] Bahareh Goodarzi, Farzad Khorasani, Vivek Sarkar, and Dhrubajyoti Goswami. 2019. High Performance Multilevel Graph Partitioning on GPU. In *2019 International Conference on High Performance Computing and Simulation (HPCS)*. 769–778. https://doi.org/10.1109/HPCS48598.2019.9188120

[29] Oded Green and David A. Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *HPEC*. IEEE, 1–6.

[30] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD Conference*. ACM, 1067–1082.

[31] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.* 3 (2005), 158–182. https://api.semanticscholar.org/CorpusID:9754025

[32] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. *Proceedings of the 2018 International Conference on Management of Data* (2018). https://api.semanticscholar.org/CorpusID:44119881

[33] Tayler H Hetherington, Timothy G Rogers, Lisa Hsu, Mike O'Connor, and Tor M Aamodt. 2012. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 88–98.

[34] Lin Hu, Lei Zou, and Yu Liu. 2021. Accelerating Triangle Counting on GPU. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 736–748. https://doi.org/10.1145/3448016.3452815

[35] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: general-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In *SC*. IEEE/ACM, 72.

[36] John Ingraham, Vikas K. Garg, Regina Barzilay, and Tommi S. Jaakkola. 2019. Generative Models for Graph-Based Protein Design. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 15794–15805.

[37] Anand Padmanabha Iyer, Erran L. Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *International Workshop on Graph Data Management Experiences and Systems*.

[38] Wuyang Ju, Jianxin Li, Weiren Yu, and Richong Zhang. 2016. iGraph: an incremental data processing system for dynamic graph. *Frontiers of Computer Science* 10 (2016), 462–476.

[39] Seunghwa Kang, Joseph Nke, and Brad Rees. 2022. Analyzing Multi-trillion Edge Graphs on Large GPU Clusters: A Case Study with PageRank. In *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*. IEEE, 1–7. https://doi.org/10.1109/HPEC55821.2022.9926341

[40] Seunghwa Kang, Joseph Nke, and Brad Rees. 2022. Analyzing Multi-trillion Edge Graphs on Large GPU Clusters: A Case Study with PageRank. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC55821.2022.9926341

[41] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *SIGMOD Conference*. ACM, 1695–1698.

[42] Dae Hee Kim, Rakesh Nagi, and Deming Chen. 2020. Thanos: High-Performance CPU-GPU Based Balanced Graph Partitioning Using Cross-Decomposition. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 91–96. https://doi.org/10.1109/ASP-DAC47756.2020.9045588

[43] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *USENIX Conference on File and Storage Technologies*.

[44] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14 (2021), 1053–1066.

[45] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. 2021. Dycuckoo: dynamic hash tables on gpus. In *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 744–755.

[46] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *IEEE International Conference on Data Engineering*.

[47] Claudio Martella, Roman Shaposhnik, and Dionysios Logothetis. 2015. Practical Graph Analytics with Apache Giraph. In *Apress*.

[48] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *ACM SIGOPS Oper. Syst. Rev.* 55 (2021), 21–37.

[49] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. 2022. Essentials of Parallel Graph Analytics. In *IPDPS Workshops*. IEEE, 314–317.

[50] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydın Buluç. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. *Proceedings of the 2021 International Conference on Management of Data* (2021).

[51] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. 2020. C-SAW: a framework for graph sampling and random walk on GPUs. In *SC*. IEEE/ACM, 56.

[52] David Püroja, Jack Waudby, Peter A. Boncz, and Gábor Szárnyas. 2023. The LDBC Social Network Benchmark Interactive workload v2: A transactional graph query benchmark with deep delete operations. *CoRR* abs/2307.04820 (2023).

[53] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1876–1888.

[54] Dipanjan Sengupta and Shuaiwen Song. 2017. EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU. In *Information Security Conference*.

[55] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey S. Young, Matthew Wolf, and Karsten Schwan. 2016. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *European Conference on Parallel Processing*.

[56] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (2017), 107–120.

[57] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate Streaming Graph Analysis through Preprocessing Buffered Updates. *Proceedings of the ACM Symposium on Cloud Computing* (2018).

[58] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* 50, 6 (2018), 81:1–81:35.

[59] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*. ACM, 135–146.

[60] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1887–1901.

[61] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. 2014. Towards large-scale graph stream processing platform. *Proceedings of the 23rd International Conference on World Wide Web* (2014).

[62] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *PPoPP*. ACM, 11:1–11:12.

[63] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *SIGMOD Conference*. ACM, 1813–1828.

[64] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[65] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *SC*. IEEE / ACM, 60:1–60:13.

[66] Martin Winter, Rhaleb Zayer, and Markus Steinberger. 2017. Autonomous, independent management of dynamic graphs on GPUs. *2017 IEEE High Performance Extreme Computing Conference (HPEC)* (2017), 1–7.

[67] Carl Yang, Aydin Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 48, 1 (2022), 1:1–1:51.

[68] Lyuheng Yuan, Akhlaque Ahmad, Da Yan, Jiao Han, Saugat Adhikari, Xiaodong Yu, and Yang Zhou. 2024. G^2-AIMD: A Memory-Efficient Subgraph-Centric Framework for Efficient Subgraph Finding on GPUs. In *ICDE*. IEEE, 3164–3177.

[69] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.

[70] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2019. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13 (2019), 1020–1034.

[71] Lei Zou, Fan Zhang, Yinnian Lin, and Yanpeng Yu. 2023. An Efficient Data Structure for Dynamic Graph on GPUs. *IEEE Transactions on Knowledge and Data Engineering* 35 (2023), 11051–11066. https://api.semanticscholar.org/CorpusID:244956057