Efficient Graph Data Access for Out-of-Memory GPU Streaming Graph Processing

Qiange Wang[1], Yongze Yan[2], Hongshi Tan[1], Cheng Chen[3], Cheng Zhao[3], Jiaming Tian[2], Jiangxin Jiang[1], Xiaoliang Cong[3], Yanfeng Zhang[2], Ge Yu[2], Weng-Fai Wong[1], Bingsheng He[1]
[1]National University of Singapore [2]Northeastern University [3]ByteDance Inc
{wangqg,hongshi,jiangjx,wongwf,hebs}@comp.nus.edu.sg;{yongzeyan@stumail,2301945@stu,zhangyf@mail, yuge@mail}.neu.edu.cn;{chencheng.sg, zhaocheng.127, congxiaoliang}@bytedance.com

Abstract

Leveraging GPUs' high parallelism can significantly improve the real-time computation efficiency of streaming graph processing. However, when a large-scale graph exceeds GPU memory capacity, CPU-GPU cooperative processing often results in substantial and irregular CPU-to-GPU data transfer overhead. This stems from the extensive redundant graph accesses during continuous computation, which can hardly be addressed by existing out-of-memory graph processing techniques. In this work, we present Grapin, an out-of-memory GPU streaming graph processing system designed to minimize graph data transfer via two effective techniques for eliminating redundant accesses: (1) Extending advanced incremental processing algorithms to GPUs by converting their heavyweight data dependency processing into GPU-friendly forms, thereby eliminating redundant graph accesses from the computation side; and (2) providing a lightweight yet efficient GPU hot subgraph management framework that finely reuses the frequently accessed dynamic subgraphs on the GPU in a vertex-centric manner. Experimental results demonstrate that Grapin can efficiently process large-scale streaming graphs with billions of edges on a single NVIDIA A5000 GPU. Enabling incremental computation reduces data transfer by 61%, and the integration of GPU hot subgraph reuse further reduces the remaining transfer by 72%, resulting in a total reduction of 89%. Compared with CPU-based solutions, Grapin achieves speedups ranging from 1.8x to 96.9x (17.9x on average).

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Yongze-zzz/C-GpuStreamGraph.

1 Introduction

In online services such as ByteDance's e-commerce and advertising platforms, thousands of graph analysis tasks, including breadth-first search, graph clustering, and product ranking, are executed daily on massive, dynamic graphs. These graphs undergo rapid changes, continuously modeling interactions such as likes, comments, and subscriptions between users, videos, and products (vertices). A high-performance streaming graph processing system that efficiently maintains up-to-date query results as the graph evolves [42] is necessary to ensure high-quality services for real-time recommendation and fraud detection tasks.

Streaming graph processing involves continuous computation over dynamic graphs to maintain the result of a single query over time, as opposed to *static* graph processing, which typically performs one-shot computation. To improve computational efficiency, recent research has explored leveraging GPU-accelerated processing [3, 6, 16, 37, 48, 49]. While these frameworks have demonstrated

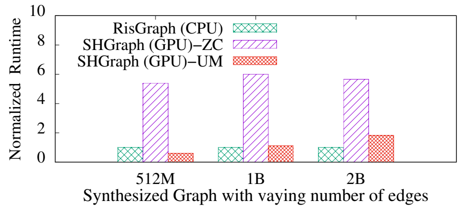


Figure 1: Performance of extending zero-copy access (ZC) and unified memory management (UM) techniques to GPU-based SSSP computation on synthetic graphs [20] with varying numbers of edges.

promising results, they rely on storing the entire graph in the GPU to exploit the massive parallelism and high memory bandwidth. They struggle to handle large graphs that cannot fit into the GPU memory, requiring the graph to be stored in CPUs and accessed subgraphs to be transferred on demand at runtime. The continuous transfer of irregular and dynamically changing graph data often leads to significant performance degradation.

Traditional out-of-memory GPU systems [9, 14, 25, 29, 30, 38, 55] typically employ sparsity-aware communication techniques based on Unified Virtual Addressing (UVA) [40], which maps CPU and GPU memory into a shared address space. These systems enable zero-copy access [29] or adopt page-centric unified virtual memory management [14, 25], allowing GPUs to seamlessly access required graph data from the CPU with high efficiency. However, in streaming graph processing scenarios, existing communication techniques demonstrate suboptimal performance, as they primarily optimize transfers of inactive data for short-duration and simple computation tasks. They lack mechanisms to eliminate the data transfer caused by duplicate graph accesses during continuous computation. Figure 1 shows the performance of directly extending zero-copy (ZC) access [29] and unified virtual memory (UM) [2] techniques to a recent in-memory GPU streaming graph processing framework, SHGraph [3]. We observe that SHGraph with ZC can be up to 6.2x slower than one of the state-of-the-art CPU-based solutions, RisGraph [12]. While SHGraph with UM improves performance on small graphs by caching and reusing most data in the GPU, its efficiency deteriorates as graph sizes increase due to frequent heavyweight memory page migrations between the CPU and GPU, ultimately lagging behind CPU-based solutions. However, developing efficient mechanisms to minimize redundant graph data accesses for streaming graph processing can be challenging.

First, there is a lack of efficient GPU computation engines capable of eliminating redundant graph accesses. In streaming graph processing, a significant portion of redundant data accesses stems from computations over already converged results. Addressing this issue typically requires incremental algorithms that track each vertex's data dependency (i.e., the parent vertex responsible for its convergence) to avoid recomputing vertices whose parent's result remains unchanged [15, 42]. However, implementing such algorithms on GPUs is challenging because correct dependency tracking requires atomically updating both the vertex result and its associated data dependency during label propagation. Unfortunately, GPUs do not provide efficient mechanisms for implementing atomic updates across multiple memory locations, limiting the practicality of incremental algorithms under massive parallelism.

Second, there is a lack of effective mechanisms for reusing redundant GPU data accesses during long-duration computations. Continuous computation in streaming graph processing presents significant opportunities to reuse frequently redundant accesses to graph data. However, existing data reuse mechanisms rely on coarse-grained, fixed-length, and heavyweight paged memory management. When handling sparse and irregular edge accesses in frequently updated graphs, this approach often migrates and caches excessive, unnecessary graph data, resulting in inefficient communication and suboptimal memory utilization [29, 45].

In this work, we present Grapin, a high-performance out-of-memory GPU streaming graph processing system that executes computations on the GPU while storing graph data in CPU memory. To minimize redundant graph accesses, Grapin integrates two key runtime redundancy elimination functions. First, it features a **redundancy-eliminating incremental computation engine**. The engine enables advanced incremental algorithms under massive GPU parallelism [15, 42] by decoupling the atomic update of the converged result and its dependency data into a sequence of independent GPU-native compare-and-swap (CAS) operations. This reduces redundant graph accesses from the computation side while maintaining compatibility with well-established vertex-centric GPU graph processing techniques. Second, Grapin provides a **lightweight GPU hot subgraph management framework**. This framework finely tracks the frequently accessed subgraphs in a vertex-centric manner while storing them compactly in the GPU memory to maximize data reuse. Through snapshot-oriented data replacement, vertex-centric edge data migration, and chunked memory management, Grapin minimizes the hot subgraph maintaining overhead on the GPU. Additionally, Grapin adopts a GPU-optimized data structure that improves data placement, enabling efficient access to dynamic graphs without compromising update performance.

Experiments on an NVIDIA A5000 GPU demonstrate that Grapin reduces graph accesses by 28%-71% (avg. 61%) through its redundancy-eliminated computation engine and further decreases CPU-GPU data communication by 67%-80% (avg. 72%) by efficiently caching and reusing the frequently accessed subgraph on the GPU. These two techniques bring a total of 89% transfer reduction. Overall, Grapin achieves speedups ranging from 1.8x to 96.9x (avg. 17.9x) over CPU-based systems. Furthermore, we demonstrate the effectiveness of Grapin on real-world graphs from our industrial partner.

2 BACKGROUND

2.1 Streaming Graph Processing

Let $G = (V, E)$ be a directed or undirected graph, where V is a set of vertices and $E \subset V \times V$ is a set of edges. A graph algorithm is defined by a function A, which is iteratively executed over the graph. We denote the result after convergence as $S' = A^{\infty}(G, S^0)$, where

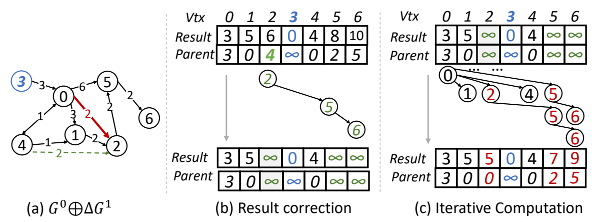


Figure 2: Incremental SSSP computation trace on a toy graph starting from vertex 3. Thick red arrows indicate edge additions and dashed green arrows indicate edge deletions.

$S' = A(G, S')$. In streaming graph processing, the algorithm A operates on a graph whose structure is modified by a stream of updates, including edge and vertex additions and deletions. To maintain consistency, updates are batched into a sequence of $\{\Delta G^0, \Delta G^1, \ldots\}$. The goal is to compute the latest result for each snapshot G^i, i.e., $S^i = A^{\infty}(G^i, S^i)$, where $G^i = G^{i-1} \cup \Delta G^i$. The naive approach of recomputing S^i from the initial state S^0 is inefficient as it reprocesses many already converged vertices for every snapshot [15, 42].

Incremental computation with dependency-memoization. Recent research has explored *Dependency-Memoization* (DM) techniques [36, 42] that utilize cached data dependencies to bypass the converged vertices and reduce unnecessary computation (i.e., incremental streaming graph processing [42]). In the DM algorithm, data dependencies indicate the direct parents on the critical computation path that leads to convergence. The DM technique decomposes computation into two stages: *result correction* and *iterative calculation*, allowing it to identify and recompute the vertices affected by graph changes separately. Figure 2 shows an example of using DM on the SSSP algorithm. The *Result* array stores the shortest paths, while the *Parent* array stores the direct predecessor on the shortest path (i.e., data dependency). In the *result correction* stage, an invalid message propagates from all affected vertices to their outgoing neighbors. Each vertex receiving the invalid message from its predecessor (indicating that its shortest path passes through the affected vertices) marks its result as invalid and propagates the invalid message to its neighbors. The *iterative computation* stage begins with the active vertices generated by *result correction* and iteratively processes all vertices until convergence. We summarize related work in Section 8 and refer interested readers to [15, 21, 42].

Streaming graph processing on the GPU. Extending the Dependency Memoization algorithm to the GPU is a challenging task because it requires maintaining the consistency of *Result* and *Parent* for every vertex during *iterative computation*, i.e., modifying the *Parent* to the direct precursor as soon as the *Result* is updated to ensure correctness. CPU-based solutions typically rely on exclusive locks to synchronize the states [12, 15] across *Result-Parent* pairs. However, GPUs lack such effective mechanisms to address the issue. The native CAS operations provided by CUDA [18] and OpenCL [33] support only a single value of basic data types. The low branching capability and massive parallelism make implementing CPU-like exclusive locks uneconomical [5]. Consequently, GPU-based streaming graph systems often resort to a naive approach [3, 6, 16, 37, 48, 49] that *recomputes from scratch*. Although the high memory bandwidth and massive parallelism help mitigate the overhead of redundant computation, these systems still struggle to process large graphs due to limited GPU memory capacity.

Table 1: The access volume on the five graphs when running SSSP with 10 batches of 100K edge mutations.

	#Edge	Volume of edge access					Changed
		Total	Total	Redun	Redun	#Top20%	edge volume
		w/o DM	with DM	#Intra	#Inter	Vertices	in each batch
OK	0.10B	4.76B	0.99B	0.12B	0.78B	0.54B	16.8M
WK	0.41B	9.12B	4.59B	0.62B	3.58B	3.5B	16.3M
TW	1.83B	22.79B	18.64B	2.11B	14.72B	3.70B	19.7M
FR	2.41B	88.29B	30.02B	5.46B	22.15B	11.21B	19.5M
UK	3.07B	157.12B	50.72B	22.89B	25.09B	22.45B	16.4M

Redundant graph accesses among long-duration computations. Streaming graph processing continuously updates the result of a query over time. This causes many vertices to be accessed repeatedly, even when employing advanced incremental algorithms [15]. Figure 3 (a)-(d) shows the DM computation traces for the SSSP algorithm on an example graph (with two batches of updates, ΔG^1 and ΔG^2), where redundant accesses occur both within and across batches, causing increased communication overhead.

To demonstrate the practical impact of redundant access, we evaluate the edge data access volume of running SSSP with 10 batches in Table 1. The results show that the DM algorithm reduces 20%-79% data transfer compared to the recomputation-based approach, which is essential for high performance. However, the total access volume remains high, ranging from 9.9x to 16.5x the number of edges, with intra-batch and inter-batch redundant accesses accounting for 12%-45% and 49%-78% of the total accesses, respectively. Due to the power-law distribution, certain frequently accessed vertices exhibit higher access frequencies than others. The top 20% of frequently accessed vertices are accessed 1.3x to 3.5x more frequently than the remaining 80%, contributing an average of 47% (up to 76%) of the total edge access. This creates substantial optimization opportunities for reusing the transferred edge data.

Nevertheless, efficiently achieving this goal in streaming graph scenarios remains a challenging task. The difficulty arises not only from the sparsity of edge data access but also from access amplification triggered by updates on evolving graphs. In particular, modifying a single edge can affect the data of an entire neighborhood. As shown in Table 1, updating a small batch of edges (100K) can invalidate data volumes exceeding 100 times their original size.

2.2 Out-of-Memory GPU Graph Processing

Recent studies have proposed out-of-memory GPU graph processing that stores small-scale vertex result values (e.g., PageRank scores) and index data in the GPU while placing large-scale edge data, such as neighbor IDs and edge weights, in the CPU [12, 29, 45]. During iterative computation, the computation engine efficiently accesses all vertex data stored in GPU memory, while the edge data corresponding to accessed vertices is transferred to the GPU on demand. This GPU-accelerated processing mechanism preserves the semantics and convergence guarantees of iterative graph processing [45, 57]. The main bottleneck arises from the extensive edge data communication between CPU and GPU. Unified virtual addressing (UVA) technology [29], which enables GPUs to access CPU memory directly, has become a promising solution.

Sparsity-aware communication based on zero-copy access. Whether in static or dynamic graph processing, each iteration only accesses a small portion of the randomly distributed graph data. This requires the communication method to be aware of and exploit this sparsity [12, 29, 45]. EMOGI [29] employs the zero-copy (ZC) access mechanism, allowing GPUs to access the graph data from the CPU directly. With the zero-copy mechanism, GPUs can access variable-length neighborhoods using several fine-grained and low-cost PCIe requests (each ranging from 32B to 128B) and skip unnecessary ones. This makes zero-copy technology a promising communication method for streaming graph processing. However, zero-copy access does not support data caching. As shown in Figure 3 (e), each neighborhood access requires a separate CPU-to-GPU data transfer, meaning that zero-copy access causes redundant graph transfers. In addition to zero-copy access, some frameworks [34, 57] propose mitigating access sparsity through CPU-assisted data compaction. However, these methods require additional engineering effort and often suffer from unstable communication performance due to the limited parallelism of CPUs [30, 45]. Given these limitations, we choose to adopt a GPU-centric solution.

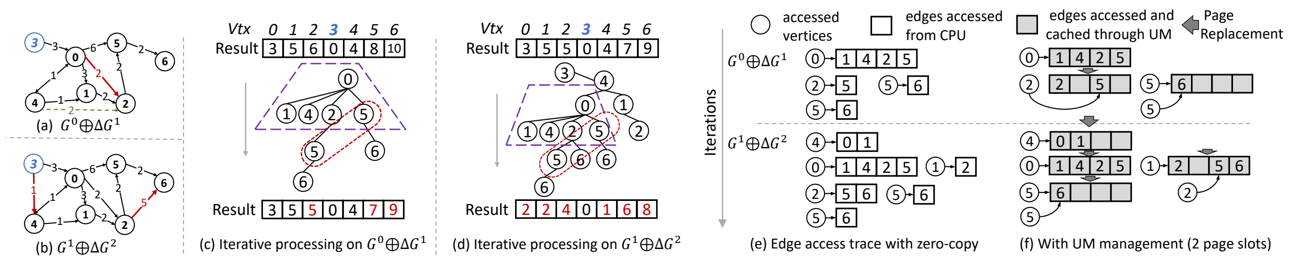
Table 2: Read and write amplification of UM management.

	OK	WK	TW	FS	UK
(a) Data transfer of SSSP computation based on a CSR[1]					
Edge accesses (GB)	2.08	13.6	24.8	58.1	127.2
Actual Transfer(GB)[2]	1.43	6.83	142.7	524.8	884.1
Read amplification	0.7X	0.5X	5.8X	9.0X	7.0X
(b) Updated pages with 100K edge mutations in VCSR [17].					
Updated pages(×4KB)[2]	118K	233K	581K	443K	440K
Updated adjlists(×8B)	16.8M	16.3M	19.7M	19.5M	16.4M
Write Amplification	3.5X	7.1X	14.7X	11.4X	13.5X

[1] We use static graph processing on the CSR to exclude the impact of sparse dynamic graph structures.
[2] The number is collected using the NVIDIA Nsight system [32].

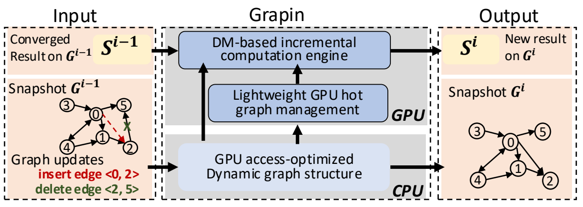
Unified memory management-based GPU data reuse. Halo [14], Liberator [14, 38], and Grus [44] store graph data in a managed memory space visible to both CPU and GPU and leverage unified memory management to support automated edge data migration and caching with a default page size of 4 KB. Figure 3 (f) shows the communication trace with unified memory management, where repeated accesses to the loaded edge pages are handled within the GPU memory (e.g., v5 in the 3rd and 7th iterations; v2 in the 5th iteration). However, the page-centric data caching mechanism is not as efficient as expected. On one hand, the heavyweight TLB invalidation causes each page fault handling to take tens of microseconds, causing low PCIe bandwidth utilization [29, 34]. On the other hand, the coarse-grained page migration and update-triggered page invalidation can lead to a significant amount of unnecessary data transfer and memory consumption, far exceeding the amount of accessed data. We evaluate this impact in Table 2 by measuring the data access volume for graphs with varying sizes. For large graphs where the data size exceeds the GPU memory capacity, page transfer volume can be 9.0x larger than edge accesses. Furthermore, topological structure updates with streaming graphs introduce cascading memory page invalidations. As indicated by Table 2 (b), applying 100K edge mutations to a CSR-like dynamic graph structure [17] leads to 1.2x-5.8x more page faults, with only 6.8%-28.4% of the data being part of the changed subgraph. While reducing the page size is a potential solution, it is hard to implement in practice, as page size is typically determined by the operating system rather than by users or application developers.

To optimize graph data reuse in heterogeneous memory systems, recent studies, e.g., CoreGraph [19], propose preprocessing the



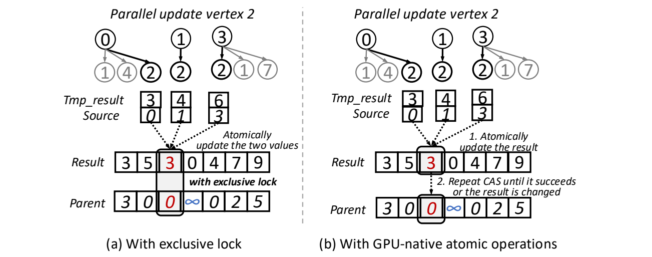
(a) $G^0 \oplus \Delta G^1$
(b) $G^1 \oplus \Delta G^2$
(c) Iterative processing on $G^0 \oplus \Delta G^1$
(d) Iterative processing on $G^1 \oplus \Delta G^2$
(e) Edge access trace with zero-copy
(f) With UM management (2 page slots)
accessed vertices
edges accessed from CPU
edges accessed and cached through UM
Page Replacement
Iterations
Vtx
Result

Figure 3: (a)-(d) shows SSSP computation on an example graph (G^0) with two batches of updates (G^1 and G^2). Zero-copy mechanism (e) directly accesses the graph data from CPU in a vertex-centric manner, its transfer volume is almost equal to the edge access volume (transferring 18 edges in the example). Unified memory management (f) migrates and caches memory pages containing accessed graph data, the data transferred can far exceed the actual demand (transferring 28 edges in the example) due to coarse-grained page replacement.



Input
Grapin
Output
Converged Result on G^{i-1}
S^{i-1}
Snapshot G^{i-1}
Graph updates
insert edge <0, 2>
delete edge <2, 5>
DM-based incremental computation engine
Lightweight GPU hot graph management
GPU
GPU access-optimized Dynamic graph structure
CPU
S^i
New result on G^i
Snapshot G^i

Figure 4: Grapin overview.



Parallel update vertex 2
Tmp_result
Source
Atomically update the two values
Result
with exclusive lock
Parent
1. Atomically update the result
2. Repeat CAS until it succeeds or the result is changed
(a) With exclusive lock
(b) With GPU-native atomic operations

Figure 5: A graphical comparison of result-dependency atomic update approaches in CPU-based systems (a) and Grapin (b), illustrating how multiple vertices concurrently update the shortest path.

graph to extract a frequently accessed core structure to accelerate computation. However, these approaches rely on heavyweight offline processes that run graph algorithms (e.g., BFS or SSSP) multiple times on the input graph. Therefore, they are impractical for streaming graphs, as rebuilding the core subgraph for every snapshot (to ensure correctness) would incur substantial overhead, often exceeding the cost of the computation itself (as discussed in Section 5.1). Another line of work improves page-based data reuse by reorganizing the graph layout or regularizing data access patterns to enhance locality [14, 31, 56]. However, these approaches remain impractical due to the high overhead of reorganizing the entire graph structure in real time. Moreover, they still suffer from inefficient communication, as evidenced by the comparison against unified memory-based approaches in Section 7. In summary, developing effective data reuse mechanisms for long-duration streaming graph processing is essential but challenging. It requires efficiently handling the sparse graph access patterns, maximizing cache utilization, and detecting graph changes with low overhead.

3 Grapin System

We present Grapin, an out-of-memory GPU system designed for high-performance streaming graph processing. An overview is shown in Figure 4. Grapin maintains the dynamic graph in CPU memory while storing vertex data and performing computation on the GPU. It leverages zero-copy access to enable transparent and efficient on-demand retrieval of the edge data of accessed (unconverged) vertices from CPU memory during computation. This allows Grapin to preserve the semantics and convergence of iterative graph algorithms. Grapin minimizes CPU-GPU memory access through two key redundancy elimination components.

DM-based incremental computation engine. To minimize redundant accesses caused by computing converged vertices, Grapin extends the advanced DM algorithm to GPUs (Section 4). It introduces a decoupled updating mechanism that transforms the heavyweight atomic update of result-parent pairs into a sequence of independent GPU-native CAS operations, seamlessly integrating the DM algorithm into vertex-centric GPU computation engines.

Algorithm 1 IncCompNbr(Tvtx src, Tvtx dst, Tval ewght, Tvtx *parent, Tval *result, Tval *buffer)

```
1: new_path = buffer[src] + ewght; //(Line 1-2) update the result
2: old_path = atomicMin(&result[dst], new_path)
3: if new_path<old_path then //(Line 4-7) update the dependency
4:     old_p=parent[dst]
5:     repeat
6:         old_p= atomicCAS(&parent[dst],old_p,src)
7:     until (new_path!=result[dst]||src==parent[dst])
```

GPU hot (frequently accessed) graph management. To minimize redundant accesses in long-duration computations, Grapin introduces a lightweight yet efficient GPU hot graph management framework (Section 5). This framework tracks frequently accessed subgraphs at the vertex level and compactly stores them in GPU memory for fine-grained graph reuse, enabling Grapin to access only cold data from the CPU, thereby reducing data transfers. To ensure the correctness of iterative processing, Grapin maintains the latest snapshot by applying all updates to the CPU graph storage and synchronizes updates to the GPU hot subgraph cache for every graph snapshot through an efficient snapshot-oriented data replacement module, which leverages GPU-parallel, vertex-centric hot subgraph tracking and edge data loading, along with chunked memory management, to minimize data replacement overhead.

4 DM-based Incremental Computation on GPUs

As described in Section 2.1, the DM algorithm decomposes computation into two phases. The *Result correction* phase performs a BFS traversal from the initially updated vertices, identifying those requiring recomputation by checking whether the current result's *Parent* depends on the affected vertices. This phase can be naturally embedded into existing vertex-centric GPU computation kernels

[43, 47]. In contrast, the *iterative computation* phase iteratively recomputes the *Result* values for those checked in the previous stage and updates the parent value accordingly. In this phase, the result value and dependency data (i.e., *Result* and *Parent*) must be atomically updated under massive parallelism during computation. However, the lack of atomicity support for values spanning multiple memory locations prevents GPUs from supporting such operations (as discussed in Section 2.1). To address this limitation and enable incremental processing, Grapin introduces a result-dependency decoupled update mechanism.

The core idea of our approach is to decouple an atomic *Result-Parent* update into a sequence of independent CAS operations on each of them. We observe that for certain iterative graph algorithms, the value of each vertex is typically updated monotonically by an accumulative vertex update function (e.g., SSSP using the `min` operation [46, 53]). This monotonicity ensures that each *Result* value eventually converges to a deterministic optimum. As a result, the *Parent* value only needs to point to the final optimal value selected by the vertex update function, rather than recording every intermediate state during computation. Based on this observation, we adopt a decoupled update approach, as illustrated in Figure 5 (b). First, concurrent threads atomically update *Result* using atomic vertex update operations such as `min()` and `max()`. Next, threads that successfully update *Result* attempt to compete for the optimal *Parent* by modifying it through a loop of compare-and-swap operations. The loop exits if: 1) The update succeeds; or 2) The *Result* value changes, indicating a better value is found. In case 2), the intermediate *Parent* value can be discarded early to reduce overhead.

Compared to CPU-based approaches that use heavyweight exclusive locks (as shown in Figure 5 (a)), Grapin requires only a series of lightweight CAS operations on basic data types, making it highly suitable for GPU's massive parallelism. In addition, the decoupled update method preserves the integrity of the vertex-centric graph processing model. This allows seamless integration with existing, highly optimized graph processing engines without requiring modifications to the original data structure, computation logic, or APIs. Grapin extends a state-of-the-art GPU graph processing system, SEP-Graph [43], and provides a similar vertex-centric programming interface. Algorithm 1 shows the neighbor processing function of the SSSP algorithm. The function begins with an atomic shortest path update on the *Result* (Lines 1-2), which is the same as the operations in SEP-Graph. Subsequently, the *Parent* update (Lines 4-7) is performed for vertices successfully updated in Line 3 via a loop of CAS operations. The loop terminates when the update succeeds (if `src==parent[dst]`) or a shorter path that invalidates the current parent is found (`new_path!=result[dst]`). The data dependency handling process (Lines 4-7) is algorithm-agnostic. Users do not need to reimplement dependency handling for each algorithm.

Correctness. The DM-based algorithm with the decoupled update implementation requires input graph algorithms to satisfy the monotonicity [15, 42], ensuring that under concurrent CAS operations with multiple competing values, each vertex's *Result* deterministically converges to an optimal value (e.g., the shortest path), regardless of the update order [53]. In the original design, a single atomic update, i.e., **comparing** the *Result* **and swapping** both the *Result* and *Parent* fields, ensures that the *Parent* always points to the current *Result* until an optimal *Result* is reached. For the decoupled approach, correctness can be ensured by proving that the optimal *Result* and optimal *Parent* (produced by the optimal *Result*) can be obtained separately after concurrent execution. Since the algorithm is monotonic, parallel CAS on the *Result* field intuitively converges to the optimal value. In the following, we enumerate all possible execution interleavings across threads to demonstrate that the *Parent* field eventually reaches the optimal value. We observe that, after the *Result* updates complete, threads fall into one of three possible cases:

-**Case 1. Failed *Result* update:** If a thread fails to update the *Result*, it implies the proposed value is not optimal. Consequently, it will not enter the loop and will make no changes to the *Parent* value.

-**Case 2. Successful *Result* update with sub-optimal values:** The looped CAS on *Parent* in Lines 5–7 results in two sub-cases: 2.1) The *Parent* update temporarily succeeds. However, the *Parent* value will eventually be overwritten by a better *Result*, which must appear later since the current *Result* is sub-optimal. 2.2) A better *Result* appears before the *Parent* is updated, terminating the looped CAS and discarding the current *Parent* update (Line 7).

-**Case 3. Successful *Result* update with the optimal value:** If a thread successfully updates the *Result* with the optimal value, the looped CAS in Lines 5–7 of Algorithm 1 will eventually update the *Parent* to the optimum based on the current *Result*, as no better *Result* value will appear to override the loop.

In all scenarios, only the *Parent* of the optimal *Result* is retained. This guarantees the correctness of the decoupled update mechanism under parallel execution.

Algorithm 2 Scheduling for a set of active vertices V_{act}

```
1: for each $v \in V_{act}$ do: assign a warp or block based on CTA [24]
2:     if warp_id==0 then
3:         {adj_s, adj_e, flg} ←.Grapin.adj_index ($v$)
4:     broadcast(adj_s, adj_e, flg) //within the warp/block
5:     for offset from adj_s to adj_e do //in parallel
6:         {$u$, $w_{v,u}$} ← Grapin.adj_list(offset,flg)
7:         IncCompNbr($v$, $u$, $w_{v,u}$ . . . ) //computation code
```

Condition checking. Checking whether a graph algorithm is monotonic is a key step in enabling incremental computation. Fortunately, recent studies have investigated sufficient conditions for monotonicity based on the property of the edge message and vertex update functions in the label propagation [11, 15, 46]. They also provide automated condition checkers based on satisfiability modulo theories (SMT) solvers [15]. Users can directly leverage these tools to verify whether an algorithm can be deployed in Grapin.

Load-balanced vertex scheduling with unified graph access. Grapin's computation engine builds on recent advances in GPU graph processing [4, 43], using Cooperative Thread Array scheduling [24] to adaptively assign each vertex to a warp or block and coalesce edge accesses for load balancing, as shown in Algorithm 2. Since Grapin manages graph data with both CPU and GPUs, during computation, all threads within a warp/block use a leading thread to obtain the start and end positions, along with a flag `flg` indicating whether the data originates from the CPU or GPU. Then, Grapin allows each warp of threads to access the adjacency list in parallel, where accesses to the CPU graph are handled via zero-copy access.

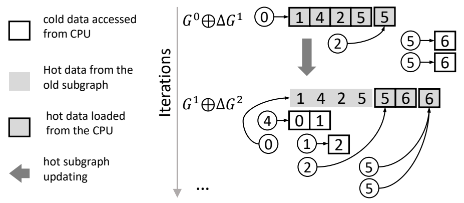

cold data accessed from CPU
Hot data from the old subgraph
hot data loaded from the CPU
hot subgraph updating
Iterations
$G^0 \oplus \Delta G^1$
$G^1 \oplus \Delta G^2$
...

Figure 6: The hot subgraph is stored compactly in GPUs and replaced in snapshots. Data migration is carry out in a vertex centric manner, guaranteeing low overhead.

5 Lightweight GPU hot graph Management

As discussed in Section 2.2, efficiently maintaining the frequently accessed subgraph for streaming graphs requires *precise and efficient tracking* of sparsely accessed vertices, *memory-efficient cache organization*, and *low-cost data replacement*. To accurately and efficiently maintain the evolving frequent-accessed subgraphs, we define the frequently accessed subgraph at the granularity of a snapshot, i.e., $HotSG^{i-1}$, and model the cache management problem as the efficient storage and transition of $HotSG$s across snapshots.

Problem statement. For each graph snapshot $G^i = (V^i, E^i)$, the frequently accessed subgraph $HSG^i = (HV^i, HE^i)$ is defined as the subgraph induced by a selected set of hot vertices $HV^i \subseteq V^i$ and their outgoing edges in G^i. This subgraph must satisfy the following three conditions. First, *hotness ranking*: for all $v \in HV^i$ and $u \in V^i \setminus HV^i$, we require that $hotness(v) \geq hotness(u)$, ensuring that all vertices included in $HotSG^i$ are more frequently accessed than those excluded from it. Second, *edge data consistency*: for each $v \in HV^i$, all of its outgoing edges in G^i must be preserved in $HotSG^i$, that is, $\{\langle v,u\rangle \in E^i\} = \{\langle v,u\rangle \in HE^i\}$. This ensures that the edge data in the $HotSG^i$ remains identical to that in the most recent graph snapshot, thereby avoiding correctness issues caused by accessing inconsistent neighborhoods. Third, *memory constraint*: the total size of $HotSG^i$ must not exceed a user-specified budget based on the remaining available GPU memory. The objective of GPU hot graph management is to efficiently store each $HotSG^i$ and transition from $HotSG^i$ to $HotSG^{i+1}$ when switching snapshots.

Design outline. Grapin provides three key components to support efficient GPU dynamic graph management. First, Grapin adopts a vertex-centric approach for precisely identifying subgraphs needing replacement based on historical vertex access frequency (Section 5.1). Second, to maximize memory utilization of the cache, it employs the Compressed Sparse Row (CSR) format for efficient and compact data storage (Section 5.2). Third, Grapin adopts a snapshot-oriented cache replacement framework (Section 5.3), which partitions the CSR edge array into multiple logical chunks to reduce global data movement, and leverages GPU-parallel, vertex-centric cache loading to accelerate data replacement. This design strikes a balance between cache quality and data replacement efficiency. First, vertex-centric hot subgraph tracking ensures high-quality data loading and storage for the sparse graph. Second, snapshot-oriented cache replacement leverages the GPU's high parallelism, thus eliminating the high overhead of manipulating variable-length and fragmented edge data for different vertices. Figure 6 shows the workflow. The neighborhoods of multiple vertices are stored in a compact format to save memory. The cache is updated when switching batches. In the first batch, the frequently accessed subgraph includes vertices 0 and 2, along with all their neighbors. Before the second batch of computation starts, Grapin identifies the subgraph that needs updating based on past access patterns and performs parallel replacement. It retains the neighbors of hot vertex 1 (v1, v4, v2, v5), loads the new hot neighbor of vertex 5 (v6), and updates the neighbors of hot vertex 2 due to structural changes (v5, v6). This design simultaneously enables efficient sparse data access and fine-grained data reuse.

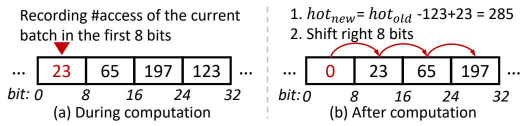


Figure 7: An example of memory-efficient hotness tracking with the window size configured to 3.

5.1 Vertex-centric Hot Subgraph Tracking

In Grapin, we compute the vertex-centric hotness score based on the past access frequency within a sliding window:

$$hotness(v)_{i+1} = \sum_{k=i-\tau}^{i} C_k(v), \tag{1}$$

where $i+1$ represents the currently scheduled batch, $C_k(v)$ denotes the access frequency (AFQ) of vertex v in batch k, and τ is the sliding window size that determines the timeliness of vertex hotness. This approach enables smooth detection of temporal variations in access frequency while mitigating the impact of sharp fluctuations.

Effectively tracking the hotness score on the GPU requires careful optimization as it involves maintaining the AFQ of all vertices in the recent τ batches. This requires additional memory of size $|V| * (\tau+1)$. We observe that the AFQ is often in the range of tens to a few hundred in graph processing, much smaller than the representation range of commonly used data types, e.g., the 32-bit `int`. Based on this observation, Grapin utilizes a single byte to track the AFQ of a vertex in each batch and calculates the hotness incrementally through bit shifting. Figure 7 shows an example with a configuration of τ=3. In this example, a 4-byte integer is used to maintain the AFQs of each vertex in 3 batches, and 2 additional bytes are used to maintain the hotness value. During computation, Grapin uses the leftmost byte to record the accesses to the vertex of the current batch. After computation, Grapin incrementally computes the hotness by subtracting the AFQ of the oldest batch and adding the AFQ of the just finished batch. Finally, Grapin shifts the integer to the right by 8 bits, removing the oldest AFQ in the rightmost byte and setting the leftmost byte to zero for future computation. In this work, we set $\tau = 3$ as the fixed parameter, as configuring τ to small values (e.g., from 3 to 6) yields similar performance, but $\tau = 3$ requires only a single integer for each vertex.

Candidate determination. According to the definition of $HotSG$, Grapin uses the top K hot vertices whose aggregated neighbor size does not exceed a given capacity as candidates for constructing the hot subgraph. First, Grapin sorts all vertices based on their hotness. Subsequently, Grapin computes the prefix sum of the degree of the sorted vertices to determine the storage requirement. Finally, Grapin employs binary search to determine the top K hot vertices whose aggregated size is smaller than a given capacity. All these operations are performed on the GPU. Thrust [8] primitives (`BlockRadixSort()` and `BlockScan()`) are used to implement high-performance sorting and prefix sum operations.

Overhead analysis. In Grapin, tracking the frequently accessed subgraph involves two lightweight steps: (1) vertex-centric hotness computation; and (2) candidate selection based on vertex sorting. The overall time complexity is dominated by the sorting stage, i.e., $O(V \log V)$, and both steps are efficiently accelerated within the GPU for real-time identification. In contrast, preprocessing-based methods [19] incur significantly higher overhead due to the heavy-weight offline processing. As a general example, CoreGraph [19] identifies the core structure by running the SSSP algorithm from M selected vertices to compute edge centrality, resulting in a total time complexity of $O(M(V+E)\log V)$, where $O((V+E)\log V)$ is the cost of a single SSSP execution. This process incurs the overhead of M out-of-memory recomputations over the entire graph, which is M times more costly than the incremental computation for each batch. Such substantial cost renders these methods impractical for streaming graph workloads.

5.2 CSR-based GPU Cache Management

Grapin uses the CSR structure to store cached subgraphs on the GPU, providing optimal memory utilization. To efficiently index the data for fast GPU retrieval, Grapin utilizes two index arrays of size $|V|$ to maintain the start position and length of the neighborhood for each vertex. For vertices outside the cache, the start position is marked invalid (-1), and the length is set to 0. This data organization enables quick verification of whether a vertex is in the cache and allows locating its edge data with $O(1)$ overhead. However, compacted data storage introduces challenges for replacing vertex neighborhoods, as it requires reorganizing the entire CSR structure to reclaim spaces for variable-length entries. This leads to substantial memory manipulation overhead. To address this, we propose a chunk-based CSR memory management technique that confines graph data manipulations to affected chunks, minimizing data movement while preserving read efficiency.

5.3 Snapshot-oriented Cache Replacement with Chunked Memory Management

Grapin virtually manages the CSR edge list using multiple equal-length logical chunks. During data replacement, only the chunks containing cold or modified data are evicted, reclaimed, compacted, and reused for newly loaded data, while the unchanged chunks remain intact. This design transforms the heavyweight full CSR data movement into more efficient intra-chunk and inter-chunk memory movement limited to affected chunks. Notably, this design does not physically modify the CSR structure, thereby ensuring that access performance remains unaffected. It also does not interfere with vertex-centric data migration. Edge data transfers between the computation engine, GPU hot subgraph cache, and CPU graph storage are still performed in a vertex-centric manner via global memory access and CPU-to-GPU zero-copy memory access. Grapin leverages GPUs to accelerate the data replacement process, as shown in Algorithm 3. Before starting computation, Grapin determines vertices that need to be loaded, evicted, and replaced due to structural updates (Lines 1-3) and allocates an array to record the volume of deleted data for each chunk (Line 4).

Parallel marking of eviction data. In this stage, a vertex-centric GPU kernel is launched to scan all deleted vertices (Lines 7-8), marking their indices and neighborhoods as invalid (0 for the degree and -1 for the neighborhoods) and incrementing the deletion counter for space reclaiming. The logical chunk ID p is computed by dividing the neighborhood start position by the chunk size (Line 6). To minimize data races over the deletion counter, Grapin maintains separate counters in shared memory for each SM, recording data locally and synchronizing at the end.

Localized space reclaiming for affected chunks. Grapin first identifies all affected chunks (Lines 9-10) using the deletion counter and then reclaims space within each chunk (Lines 11-12). The reclaiming process consists of two stages. First, new indices are computed based on the prefix sum of the degrees of the remaining vertices. Second, the remaining neighborhoods are compacted using the Thrust `remove_if()` primitive, which moves active data to the beginning while preserving the relative order [39]. After intra-chunk compaction, Grapin further compacts data across chunks to create continuous space for newly loaded data. Grapin iterates from the smallest to the largest chunk, slicing and redistributing the data to unfilled chunks to free the current chunk for loading

Algorithm 3 Parallel cache management for each snapshot.

```
1:  $vtx\_evict \leftarrow vtx\_cached \setminus cache\_candidate$
2:  $vtx\_load \leftarrow cache\_candidate \setminus vtx\_cached$
3:  $vtx\_rep\_upd \leftarrow cache\_candidate \cap vtx\_cached \cap vtx\_upd$
4:  $chunk\_del\_cnt$[num_chunks] = [0]
    1: Marking invalid data and recording deleted data volume for each chunk
5:  for each $v \in vtx\_evict\_hot \cup vtx\_rep\_upd$ do in parallel
6:      $p$ = chunkId($v$)
7:      mark_invalid_nbr($v, p, deg(v)$)
8:      $chunk\_del\_cnt[p] += deg(v)$
    2: Reclaiming space within and across affected chunks
9:  $chunk\_id\_sorted \leftarrow$chunk_sort_by($chunk\_del\_cnt$)
10: $p\_start = 0$; $p\_end$ = idx_last_nonzero($chunk\_sorted$)
11: for each $chk \in chunk\_id\_sorted[0 : p\_end]$ do in parallel
12:     $g\_cache[chk] \leftarrow$ compaction($g\_cache[chk]$)
13: $empty\_cache \leftarrow \{\}$
14: while $p\_start \neq p\_end$ do
15:     $chk\_head = chunk\_id\_sorted[p\_start]$
16:     $chk\_tail = chunk\_id\_sorted[p\_end]$
17:     cut a slice of $g\_cache[chk\_head]$ to fill $g\_cache[chk\_tail]$
18:     if if_full($g\_cache[chk\_tail]$) then
19:         $p\_end --$
20:     if if_empty($g\_cache[chk\_head]$) then
21:         $empty\_cache \leftarrow < chk\_head, s\_pos = 0 >$
22:         $p\_start ++$
23: $chk\_head = chunk\_id\_sorted[p\_start]$
24: $empty\_cache \leftarrow < chk\_head$, remain($g\_cache[chk\_head]$) $>$
    3: allocating the physical location for the newly inserted vertices
25: $local\_deg[|vtx\_load \cup vtx\_rep\_upd|]$ = [foreach...($deg(v)$)]
26: $logic\_idx[\,] \leftarrow$ prefix_sum($local\_deg[\,]$)
27: $chunk\_range[\,]$ = chunk($vtx\_load \cup vtx\_rep\_upd, logic\_idx[\,]$)
28: for each $v \in vtx\_load \cup vtx\_rep\_upd$ do in parallel
29:     $chk\_idx$ = chunk_idx($chunk\_range, v$)
30:     $offset = logic\_idx[v] - logic\_idx[chunk\_range[chk\_idx]]$
31:     $< phy\_chk\_id, s\_pos >= empty\_cache[chk\_idx]$
32:     $phy\_idx[vtx] = phy\_chk\_id *$chunk_size$+ s\_pos + offset$
    4: vertex-centric data loading using zero-copy access
33: for each $v \in vtx\_load \cup vtx\_rep\_upd$ do in parallel
34:     load_nbr_from_cpu(v, $phy\_idx\_start[vtx]$, $deg(v)$) //zero-copy
```

new data (Lines 14-22). The reclaimed chunks and their local start indices *s_pos* are appended to an empty cache list (Line 21) for subsequent memory allocation. The value of *s_pos* indicates the starting position of the available space of a chunk, set to zero for an empty chunk, and a non-zero value for the last chunk containing the remaining data.

Parallel space allocation. First, Grapin computes the prefix sum of vertex degrees to determine the logical index for each to-be-loaded vertex (Lines 25-26). Then, it calculates physical chunks using these logic indices, packing and slicing continuous data into chunks and assigning them to the corresponding chunk IDs, such that the aggregated neighborhood size does not exceed `chunk_size`. Grapin maintains an array to record the vertex range for each chunk (Line 27). Finally, it computes its physical indices based on the chunk ID, logical index, and available space *s_pos* of the chunk (Line 32).

Vertex-centric data loading. Grapin utilizes zero-copy access to load new data from the CPU. Each to-be-loaded vertex is assigned to a thread warp/block, with edge data loading tasks distributed across all threads. Such GPU-directed, vertex-centric data loading combined with the vertex-centric hotness computation in Section 5.1 enables Grapin to efficiently load and cache the edge data of hot vertices that are sparsely scattered across the CPU graph [29]. Application developers are relieved from explicitly gathering and transferring edge data across the graph with varying distributions.

Optimization for large-degree vertices. For hot vertices with structural changes, especially large-degree vertices, fully reloading their neighborhoods from the CPU is inefficient, as most edge data remains unchanged. To address this, Grapin transfers only the updates to the GPU, combines it with the old neighborhoods in a separate memory buffer, and writes them back to the allocated space. This optimization not only reduces data transfers but also implicitly clusters frequently updated vertices together, minimizing data movement of vertices with stable hotness.

6 Dynamic Graph Access Optimization

Optimizing CPU-to-GPU neighbor data transfers for every access is critical for performance but challenging in streaming graphs. This difficulty arises because existing dynamic graph structures often distribute graph data across non-contiguous memory segments to handle graph updates [3, 13, 16, 37, 48]. While these approaches are well-suited for in-memory computation, they incur substantial communication overhead due to remote addressing or unnecessary data transfers when the computation engine and graph structure are connected via PCIe. As shown in Figure 9, zero-copy access over existing dynamic graph structures achieves only 9% to 48% of the performance of the CSR format. To address this issue, Grapin optimizes the Packed Memory Array (PMA)-based CSR structure [17] for remote GPU access to dynamic graphs.

PMA-based CSR structure with neighbor aggregation. Figure 8 provides an overview of this design. It stores the neighborhoods of all vertices in a single PMA, ensuring compact storage for efficient single-vertex neighborhood access. Meanwhile, neighborhoods of different vertices are sparsely organized with gaps to facilitate efficient edge updates. To enable fast retrieval from GPUs, Grapin uses a pair of start and end position indices (stored in the GPU for high-performance access) to locate the edge data of each vertex.

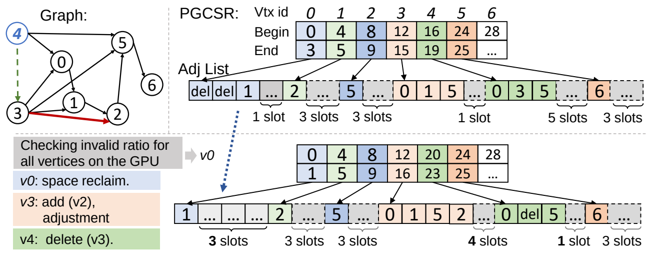


Figure 8: An example of Grapin's graph structure update on a toy graph, including inserting edge $\langle 3,2\rangle$, deleting edge $\langle 4,3\rangle$, and reclaiming space for v_0. We assume that Grapin uses a memory extension unit size of 4, meaning four slots are appended whenever the space is full. For v_0, space is reclaimed by compacting its neighborhood entries toward the head. For v_3, the new edge is inserted at the tail, followed by an adjustment that allocates four new slots at the rear. For v_4, the edge is deleted by marking it with a deletion flag.

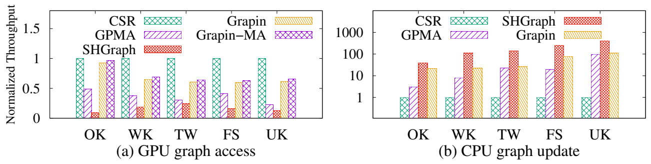


Figure 9: Throughputs of various graph data structures normalized to CSR. The experimental setup is described in Section 7. Graph access and update throughput are measured by the average number of edges processed per second during SSSP execution and the application of 10 batches of 100K edge mutations, respectively.

When there is a slot available at the end of the adjacency list, new data is appended directly, and the end position is updated. If space is insufficient, an adjustment operation is triggered, recursively checking and adjusting the positions of the neighborhoods (on the PMA array) of the "full" vertex to ensure that every vertex has adequate gaps at the end (vertex 3 in Figure 8 is an example). Grapin leverages the well-established adjustment policy [17].

Zero-copy access optimization. Grapin allocates the edge data on pinned CPU memory using the `cudaMallocHost()` function. With vertex-centric task scheduling (as discussed in Section 4), zero-copy access operates similarly to accessing the global memory [29]. Edge accesses from each warp are consolidated into PCIe memory requests and processed in a cacheline-aligned manner. If a vertex's adjacency list spans two or more cachelines, the increased memory requests can reduce data transfer efficiency [29]. To optimize this, Grapin modifies the graph structure adjustment algorithm to ensure that each vertex's starting position is a multiple of 128 bytes.

Optimization performance evaluation. Figure 9 shows the performance improvement of Grapin, without and with memory-aligned optimization (-MA), compared to three baselines: (1) CSR for static graphs, (2) GPMA [37], which adopts a naive PMA-based CSR structure without the proposed optimizations, and (3) SHGraph [3], which uses a hash-indexed list to partition and manage the adjacency list of each vertex. Both GPMA and SHGraph are state-of-the-art baselines for GPU in-memory streaming graph processing. We observe that the basic design, featuring neighbor-aggregated storage and a CSR-like data access mechanism, improves graph access throughput to 60%-93% (avg. 67.2%) of CSR. In contrast, the performance of GPMA and hash-indexed adjacency lists is only 9%-48%

Table 3: Datasets used in the experiment.

	Graphs	Vertices	Edges	Size
public	orkut (OK) [50]	2,997,166	106,350,214	1.5GB
	wiki-en (WK) [22]	13,593,032	437,208,542	6.3GB
	twitter-2009 (TW) [23]	52,579,682	1,963,263,821	28GB
	friendster (FS) [1]	68,349,466	2,586,147,869	45GB
	uk-2007 (UK) [1]	105,153,952	3,301,876,564	55GB
Industry	Production-G1	92,198,925	1,013,722,137	15GB
	Production-G2	123,297,501	2,724,318,436	42GB

of that of the CSR structure due to the need to access discontinuous edge slices. Building on the basic structure, the memory-aligned optimization (-MA) further enhances access performance by 4.3%-7.0%, achieving throughputs of 63%-97% (avg. 71.6%) of CSR. While the PMA-based CSR structure exhibits relatively lower graph update performance compared to hash-indexed adjacency lists, prioritizing access performance over update performance is a reasonable trade-off, as computations are the primary source of overhead [12].

7 Experimental Evaluation

Environments. The experiments are conducted on a GPU server equipped with two Intel Xeon Silver 4316@2.30GHz CPUs, 40 CPU cores in total, 384GB DRAM, and one NVIDIA A5000 (24GB) GPU with PCIe 4.0 interconnect. The server runs Ubuntu 20.04 OS with Linux 5.15 kernel, GCC-7.5, and CUDA 11.4.

Algorithms. The experiments involve four representative graph analysis algorithms with different computation patterns: Breadth First Search (BFS), Single Source Shortest Path (SSSP), PageRank (PR), and Connected Component (CC).

Baselines. We use three representative baselines: Ingress [15] (CPU), RisGraph [12] (CPU), and SHGraph [3] (GPU). RisGraph runs recomputation-based PageRank because it does not support incremental PageRank computation. We also implement Grapin-ZC (utilizing zero-copy access) and Grapin-Page (enabling page memory caching [35] in 4KB) as out-of-GPU processing baselines, both utilizing the dynamic graph access optimization while disabling GPU hot subgraph management. These two extensions can be considered the enhanced version of existing systems [14, 25, 29, 38, 44], featuring an efficient incremental computation engine and communication optimizations. We also extend Grapin-ZC with a recomputation-based engine (i.e., Grapin-ReComp) to demonstrate the efficiency of incremental processing. For CPU-based systems, we follow the recommended configurations [12, 15], setting 2 threads per CPU core to optimize parallel execution. For Grapin, the default GPU cache capacity is configured to 4GB. The chunk size is set to 32MB to ensure each chunk can accommodate the largest vertex while maintaining low scheduling overhead. [57]. Considering that the size of streaming graphs tends to grow beyond GPU memory, we adopt out-of-memory processing, even if the initial graph size fits into GPU memory, to better reflect practical settings.

Datasets and workloads. Table 3 presents the information on five publicly available graphs and the two product association graphs extracted from our industrial partners. For public graphs, we follow [27] to convert them into streaming graphs by uniformly selecting a set of edges from the original graph as edge updates, with 50% allocated for insertions and 50% for deletions. The updates are packed and mixed into 10 batches. This ensures that the graph updates maintain the same distribution as the original graph. For the production graphs, we use their natural timestamps to construct the workload (see Section 7.4 for details). Static graph processing is first conducted on G^0, which is constructed from edges not selected and assigned for deletion, to establish the initial fixed point. Subsequently, incremental processing is sequentially executed on the 10 batches, with each batch of updates processed until convergence. We report the time taken for the computation over all 10 batches.

7.1 Overall Comparison

We use 10 batches of 1K, 10K, and 100K edge mutations to thoroughly evaluate the effectiveness of Grapin. Table 4 shows the execution times of all systems.

Comparison with CPU-based systems. Benefiting from GPU parallel processing and the efficient dynamic graph communication, Grapin outperforms CPU-based systems across all cases. Specifically, it achieves speedups ranging from 1.8x to 96.9x (avg. 16.2x) over RisGraph and 7.4x to 70.2x (avg. 19.4x) over Ingress. RisGraph runs out of memory on the UK graph for the CC algorithm. PageRank achieves a higher speedup (avg. 33.3x) compared to BFS, SSSP, and CC (avg. 7.8x, 14.3x, and 14.0x, respectively), as its arithmetic-heavy aggregation benefits more from the GPU's parallel acceleration. The advantage of Grapin over CPU-based systems grows with graph size. Specifically, on two small graphs, Grapin achieves average speedups of 6.2x, 14.1x, 8.2x, and 14.3x for BFS, SSSP, CC, and PR, respectively. On three billion-scale graphs, Grapin achieves average speedups of 8.9x, 14.4x, 18.7x, and 45.9x for the four algorithms. Notably, the runtime depends on the proportion of changed critical paths, which does not scale linearly with batch size. This is consistent with the results in [15, 27, 42].

Comparison with GPU-based systems. While SHGraph outperforms Grapin on the smallest graph orkut, it encounters memory exhaustion issues on four larger graphs. In contrast, Grapin and its variations succeed in all cases. Grapin-Recomp fails to consistently outperform CPU-based solutions due to the high volume of redundant graph accesses to already converged data. Building upon this, Grapin-ZC, which integrates the proposed incremental computation engine, reduces data transfers by 28% to 71% (avg. 61%) and achieves performance improvements of 1.3x-2.9x (avg. 2.3x) across four algorithms. PageRank benefits less than other algorithms because computing floating-point PR values requires more access to achieve convergence, even with incremental computation. Benefiting from the efficient GPU hot subgraph management (Section 5), Grapin significantly reduces communication and achieves speedups ranging from 1.7x to 5.8x (avg. 3.0x) over Grapin-ZC. The performance improvement from hot subgraph caching remains consistent across various algorithms, yielding average improvements ranging from 2.8x to 3.3x. Moreover, the performance improvement becomes more pronounced with increasing graph scale, achieving speedups of 2.4x-2.9x on the two small graphs and 2.9x-3.5x on the three large graphs. Nevertheless, we note that even when the remaining GPU memory is insufficient to support hot subgraph management, Grapin-ZC still achieves significant speedup over CPU-based baselines. This improvement stems from the reduced data transfer overhead enabled by incremental processing.

Table 4: Execution times (in second) across 10 batches of 1K, 10K, and 100K edge mutations. OOM indicates running out of memory.

		batch size=1K					batch size=10K					batch size= 100K				
Alg.	System	OK	WK	TW	FS	UK	OK	WK	TW	FS	UK	OK	WK	TW	FS	UK
BFS	RisGraph	3.8	6.4	37.5	64.8	47.8	3.6	6.9	39.6	69.3	52.4	3.8	7.5	40.3	70.3	51.8
	Ingress	8.1	20.6	83.5	140.7	145.6	9.2	22.4	94.6	149.9	160.9	11.8	22.9	97.6	154.3	164.9
	SHGraph	0.78	OOM	OOM	OOM	OOM	**0.81**	OOM	OOM	OOM	OOM	0.74	OOM	OOM	OOM	OOM
	Grapin-RComp	5.7	11.7	31.4	79.8	160.6	5.9	12.2	31.7	83.6	163.3	6.2	12.5	33.1	85.8	176.5
	Grapin-ZC	3.7	4.9	23.1	31.6	47.1	2.9	5.0	24.1	30.2	46.2	2.6	5.5	28.1	31.9	39.3
	Grapin-Page	1.5	2.8	44.6	75.3	138.6	1.9	2.5	45.4	78.3	143.0	1.6	5.6	48.7	77.6	139.9
	Grapin	1.0	**2.1**	**6.2**	**9.8**	**14.2**	1.2	**1.9**	**7.2**	**10.4**	**14.4**	1.3	**2.5**	**8.7**	**11.0**	**14.1**
SSSP	RisGraph	16.1	17.4	44.8	221.3	135.6	13.4	17.4	53.4	195.0	140.7	15.4	21.9	61.5	199.6	148.7
	Ingress	9.5	28.8	89.4	240.0	193.9	10.3	29.0	97.4	237.8	203.0	10.9	33.4	92.2	241.2	214.8
	SHGraph	0.58	OOM	OOM	OOM	OOM	0.67	OOM	OOM	OOM	OOM	0.91	OOM	OOM	OOM	OOM
	Grapin-RComp	6.1	11.9	27.7	87.7	167.5	6.2	12.4	27.4	87.5	168.1	7.7	12.6	27.8	90.6	182.9
	Grapin-ZC	1.9	5.9	19.7	34.0	44.6	2.1	5.5	19.8	34.5	41.9	2.4	6.5	23.7	34.0	44.7
	Grapin-Page	1.8	4.4	40.3	142.4	158.8	2.4	5.1	42.2	209.2	151.9	1.6	5.0	44.7	195.2	156.2
	Grapin	0.7	**1.8**	**4.3**	**11.3**	**15.2**	0.7	**2.0**	**4.2**	**12.5**	**15.7**	1.0	**2.1**	**7.2**	**13.4**	**16.8**
CC	RisGraph	2.0	7.4	44.0	60.2	OOM	2.0	7.5	46.0	69.4	OOM	2.2	7.6	53.6	69.0	OOM
	Ingress	9.3	28.3	205.5	400.7	205.8	9.5	30.8	219.8	412.9	203.5	12.3	32.5	215.8	418.1	225.1
	SHGraph	0.64	OOM	OOM	OOM	OOM	0.88	OOM	OOM	OOM	OOM	0.79	OOM	OOM	OOM	OOM
	Grapin-ReComp	6.0	11.4	30.4	85.1	164.2	6.0	11.5	32.1	86.0	167.1	6.3	12.4	32.4	89.4	168.0
	Grapin-ZC	1.8	4.5	20.7	30.4	37.4	1.9	4.6	20.0	29.5	38.0	1.9	5.3	23.2	30.6	40.5
	Grapin-Page	1.0	2.3	40.0	80.5	74.8	1.0	2.7	43.7	73.7	67.2	1.4	4.8	43.7	81.6	75.1
	Grapin	0.8	**1.7**	**5.8**	**10.2**	**13.0**	1.1	**1.6**	**6.9**	**11.5**	**14.7**	1.1	**2.4**	**9.4**	**12.5**	**15.0**
PR	RisGraph	90.4	270.3	1986.1	2925.1	4404.7	105.8	286.1	2076.0	2930.1	4538.3	105.3	291.5	2283.0	3142.1	4720.1
	Ingress	68.2	164.2	1865.0	2119.6	2966.1	63.7	157.4	1895.4	1874.8	3664.4	67.8	197.2	2250.2	2323.2	4112.6
	SHGraph	**3.76**	OOM	OOM	OOM	OOM	**4.29**	OOM	OOM	OOM	OOM	**6.41**	OOM	OOM	OOM	OOM
	Grapin-ReComp	16.3	38.0	327.4	200.2	219.4	18.3	41.3	330.3	201.51	251.9	25.5	58.8	330.1	353.7	392.8
	Grapin-ZC	15.0	31.4	213.7	91.2	177.3	19.8	38.6	215.1	148.8	228.0	24.9	55.0	232.0	276.5	340.8
	Grapin-Page	10.8	13.6	1496.7	911.5	1103.3	11.6	15.9	1502.5	1250.8	1367.8	14.5	23.4	155.2	1709.2	1670.2
	Grapin	6.3	**12.9**	**41.0**	**30.2**	**85.3**	6.8	**13.9**	**37.3**	**89.6**	**106.7**	8.3	**14.0**	**44.0**	**91.9**	**111.3**

Table 5: The volume of communicated data (in the number of edge).

Graph	Trans-ZC	Trans-with HotSG			Trans-UM
		Total	Intra-Redun	Inter-Redun	
OK	0.99B	0.26B (↓73%)	0.04B (↓61%)	0.15B (↓81%)	0.14B(↓85%)
WK	4.6B	1.2B (↓74%)	0.3B (↓55%)	0.7B (↓82%)	2.3B (↓51%)
TW	18.6B	3.7B (↓80%)	0.8B (↓62%)	1.5B (↓90%)	38.3B (2.1x)
FS	30.0B	9.6B (↓68%)	1.8B (↓67%)	6.2B (↓72%)	167.5B (5.6x)
UK	50.7B	16.8B (↓67%)	6.5B (↓72%)	8.7B (↓65%)	151.2B (3.0x)

Table 6: Performance breakdown of Grapin-ZC (GZC) with and without hot subgraph management (HSG).

Graph	Time (s)									
	OK		WK		TW		FS		UK	
Config	GZC	+HSG	GZC	+HSG	GZC	+HSG	GZC	+HSG	GZC	+HSG
Overall	2.43	1.00	6.46	2.12	23.72	7.16	34.00	13.40	44.72	16.83
GraphUpd	0.02	0.02	0.57	0.58	2.95	2.67	0.01	0.01	0.86	0.95
HotSGRep	0	0.27	0	0.51	0	0.30	0	0.34	0	1.61
IncComp	2.41	0.69	5.89	1.04	20.77	4.19	33.99	13.05	43.86	14.27

Comparison with UM-based approaches. While Grapin-Page outperforms Grapin-ZC on two small graphs, its performance remains inferior on three large-scale graphs (excluding PR on TW with a batch size of 100K). As graph sizes increase, Grapin-Page frequently migrates memory pages containing little active data between the CPU and GPU, diminishing data reuse efficiency. In some cases (e.g., SSSP on the UK), Grapin-Page even performs worse than CPU-based systems. In contrast, leveraging fine-grained vertex-centric hot subgraph management, Grapin achieves speedups ranging from 0.9x to 40.3x (avg. 7.5x) over Grapin-Page. Grapin-Page shows comparable performance to Grapin for CC on Orkut, as the entire graph can be cached in the GPU.

Varying batch sizes. Compared to CPU-based systems and Grapin-ZC, Grapin achieves average speedups of 14.6x and 3.1x, 12.5x and 2.9x, and 12.7x and 3.1x with batch sizes of 1K, 10K, and 100K, respectively. Since its performance gains remain consistent across different batch sizes, we adopt a batch size of 100K for subsequent analysis. A detailed evaluation of Grapin 's performance under batch sizes ranging from 1 to 100M is provided in Section 7.3.

7.2 GPU Hot Graph Management Performance

Communication reduction analysis. Table 5 presents the reduction in communication volume achieved by the hot subgraph management and unified memory management using the SSSP algorithm. We observed that the proposed vertex-centric hot subgraph management leads to an overall 67%-80% reduction in CPU-GPU communication across the five graphs (Total). Specifically, the communication of redundant graph accesses within each batch (Intra-Dup) is reduced by 55%-72%, while redundant graph accesses across two sequential batches (Inter-Dup) are reduced by 65%-90%. In contrast, while unified memory exhibits good performance on small graphs by caching most data in the GPU, it fails to reduce data transfers on large graphs. Instead, its coarse-grained data migration results in a transfer volume increase of 2.1x to 5.6x.

Performance breakdown. We analyze the time cost of different components in Grapin, including GPU Computation (IncComp), CPU Graph Updating (GraphUpd), and GPU hot subgraph Replacement (HotSGRep). Communication overhead is included in IncComp, thanks to the asynchronous nature of zero-copy access. The results are shown in Table 6. Although the GPU-resident hot subgraphs significantly reduce communication overhead, incremental computation still dominates the runtime, accounting for 49.0% to 97.3% across different graphs. In contrast, on the three large graphs, HotSGRep accounts for only 2.1% to 9.6% of the total runtime, thanks to efficient snapshot-oriented data replacement. On the two small graphs, OK and WK, it contributes a higher proportion, ranging from 24.1% to 27.0% of the overall runtime. Figure 10 further shows the performance breakdown across batches. The vertex-centric hot subgraph management exhibits progressive improvement, stabilizing after three batches of warm-up. The time spent on graph updating and hot subgraph updating remains consistent across batches. For SSSP, the first batch requires more time because many critical paths are affected. Grapin can also be extended

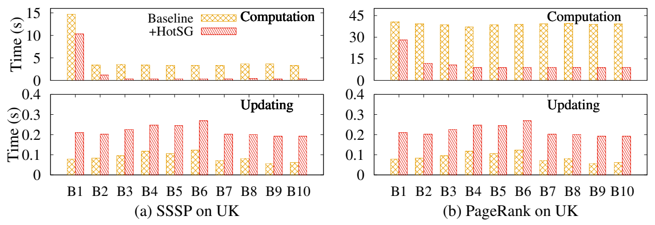


Figure 10: Performance improvement on UK across 10 batches.

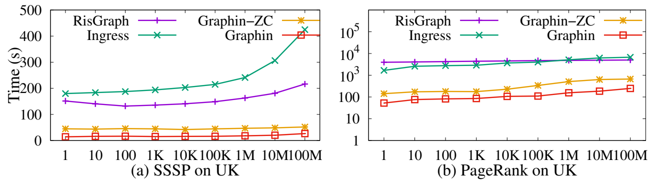


Figure 11: Performance with varying batch sizes.

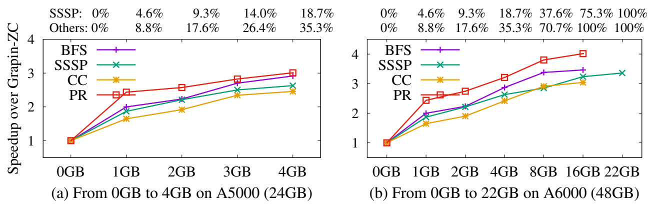


Figure 12: Performance with varying hot subgraph sizes on the UK graph on a 24GB A5000 GPU and a 48GB A6000 GPU. The upper axis indicates the ratio of cached graph for different algorithms.

to other streaming graph tasks with long-running computations, such as real-time graph random walk/sampling and continuous subgraph matching. However, developing efficient GPU engines for these algorithms is a challenging task [7, 28] and is beyond the scope of this work. We leave it for future work.

7.3 Micro Benchmark

Varying batch sizes. We vary batch sizes from 1 to 100M to evaluate the performance of Grapin in Figure 11. The runtime of PR exhibits slight growth as the batch size increases, as its computation volume is positively correlated with the number of updated edges. In contrast, the runtime of SSSP depends on whether the critical path changes and therefore exhibits non-monotonic behavior, except on Ingress. This is due to the high CSR update overhead in Ingress. Nevertheless, Grapin's effectiveness remains consistent across all cases. For the SSSP (PageRank) algorithm, Grapin achieves speedups ranging from 8.1x-10.6x (15.6x-75.1x), 11.4x-15.2x (21.0x-36.3x), and 2.6x-3.1x (2.1x-3.4x) over RisGraph, Ingress, and Grapin-ZC, respectively. When the batch size reaches 3% of the total graph size (e.g., 100M edges), most edges are affected, significantly reducing the effectiveness of inter-batch data reuse. However, Grapin still maintains a significant advantage over other baselines.

Varying the size of cached subgraphs on GPUs with different memory capacities. To evaluate the impact of hot subgraph sizes, we run all algorithms on the FS graph on an A5000 GPU with 24GB memory and an A6000 GPU with 48GB memory. On the A5000, we start with no hot subgraph caching (0GB) and linearly increase the hot subgraph size to 4GB. On the A6000, we start with no hot subgraph caching (0GB) and exponentially increase the hot subgraph size to 22GB, which is sufficient to cache all

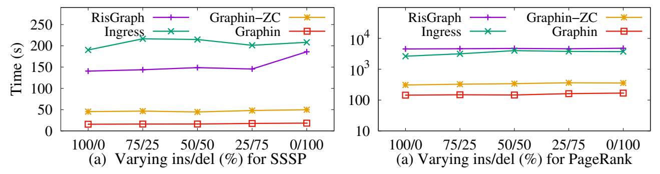


Figure 13: Performance with varying insertion and deletion ratios.

Table 7: Performance with various τ.

$\tau =$	0	1	2	3	4	5	6
SSSP	7.68	7.50	7.34	7.24	7.24	7.22	7.34
PageRank	48.31	45.94	44.60	43.96	43.82	44.37	44.41

Table 8: Runtime comparison (s) between Grapin (in-memory mode) and SHGraph under 100K edge mutations on the OK graph.

BFS		SSSP		CC		PR	
SHGraph	Grapin	SHGraph	Grapin	SHGraph	Grapin	SHGraph	Grapin
0.74	0.71	0.91	0.61	0.79	0.75	6.41	7.54

edge data. We observe from Figure 12 (a) that even with only 1GB allocated to the hot subgraph, Grapin still delivers considerable performance improvements, achieving 1.6x–2.4x speedups on the A5000 GPU. This is because frequently accessed vertices account for only a small portion of the graph (less than 10%) due to the power-law distribution. These results indicate that our approach can be adapted to GPUs of varying resource constraints. As the hot subgraph size increases from 1GB to 4GB, the per-GB benefit of caching diminishes compared to the initial 1GB. However, the overall improvement remains significant. As the cache size increases from 4 GB to 22 GB on the A6000 GPU, eventually becoming large enough to hold all edge data, the marginal benefit per additional GB decreases exponentially. As shown in Figure 12 (b), caching the entire edge set yields only a 1.2x–1.3x improvement over caching 4 GB. This is because accesses to edge data of cold vertices typically account for only a small portion. As a result, Grapin does not rely on expensive GPUs with large memory capacity. Commodity GPUs that can accommodate vertex data along with a small edge data cache can already deliver high performance and cost-effectiveness.

Varying the ratio of insertions and deletions. Figure 13 shows the performance of the four systems under varying ratios of edge insertions and deletions. We observe that Grapin consistently outperforms the other three systems. For the SSSP algorithm, RisGraph exhibits slightly inferior performance in the full-deletion workload. This is attributed to its DM incremental algorithm implementation, which incurs a high cost in correcting the result dependency tree [12]. In contrast, Grapin-ZC and Grapin employ the GPU-optimized DM implementation, demonstrating consistent performance across diverse insertion-deletion ratios.

Evaluation of τ. We evaluate the impact of τ on the runtime of SSSP and PageRank over the TW graph. As shown in Table 7, compared to disabling the sliding window ($\tau = 0$), setting $\tau = 3$ leads to a 5.7% to 9.0% reduction in overall runtime. This improvement can be attributed to effective long-term data caching, achieved by reducing data movements for vertices with unstable hotness. As τ increases from 3 to 6, the performance tends to stabilize, while the memory overhead continues to grow. Setting $\tau = 3$ achieves a reasonable trade-off between performance and memory consumption.

In-memory performance. We compare SHGraph and Grapin with in-memory processing on Orkut graph, as shown in Table 8.

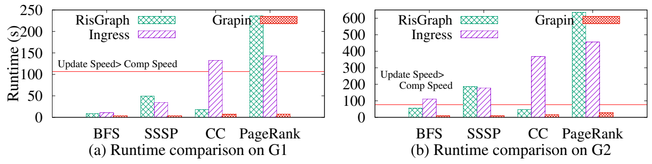


Figure 14: Performance comparison on real production graphs.

We observe that the performance gap between SHGraph and Grapin (in-memory mode) is minor, as the high memory bandwidth and massive parallelism of the GPU effectively mitigate the advantages of incremental processing. For PageRank, Grapin performs slightly worse, as the computation reduction from incremental processing on Orkut is almost negligible (as indicated by the runtime comparison between Grapin-Recomp and Grapin-ZC in Table 4), making the additional overhead difficult to amortize. Overall, Grapin is better suited for out-of-memory processing scenarios.

7.4 Case Study on Real-world Graphs

In two production graphs, vertices represent products in an online store, and edges represent product similarity captured over a 24-hour and 48-hour period, respectively. The average update frequencies are 11,733 and 15,766 edges per second for the two graphs. We pack the latest 1M edges into 10 batches of 100K mutations and feed them into RisGraph, Ingress, and Grapin. SHGraph is excluded from the comparison due to memory exhaustion issues. Figure 14 shows the evaluation across all four algorithms on a server equipped with dual Intel Xeon 8336C CPUs, 2TB of DRAM, PCIe 4.0, and an NVIDIA A100 (80GB) GPU. Grapin shows superior performance compared to CPU systems on advanced hardware. Specifically, it achieves speedups ranging from 2.0x to 22.0x (avg. 12.2x) against RisGraph and Ingress. In streaming graph applications, there are often strict requirements on the average computation time to ensure that the results can keep pace with the latest graph snapshot, preventing the delay caused by graph updates surpassing computation. In Figure 14, two red lines indicate the border of computation delay for G1 and G2, i.e., $10 * BS_{comp}/FRQ_{upd}$ $=10*100K/11733\approx85.2$ seconds and $10*100K/15766\approx63.4$ seconds, respectively. We observe that Ingress fails to meet the requirement in most cases. RisGraph fails to satisfy the requirement for SSSP on G2 and PageRank on both graphs and almost reaches the backlogging threshold for BFS and CC on the larger graph, G2. In contrast, Grapin fulfills the requirements of all workloads and retains scaling potential.

7.5 Cost-efficiency and Future Work

Compared to CPU-based solutions, Grapin offers substantial cost advantages. First, distributed processing of streaming graphs remains an open challenge due to the complexity of managing dynamic graphs [12, 15] across CPU nodes. Second, the financial cost of building a CPU cluster is higher. Even if CPU-based solutions achieved theoretically optimal scalability, processing a billion-edge graph would still require an average of 18 CPU nodes. According to Alibaba ECS [10] quotation, renting 18 CPU nodes with comparable computational power (each with 32 vCPUs and 128 GB memory at 1.0 USD/hour) incurs a total cost more than 7 times that of renting a single GPU node (32 vCPUs, 346 GB memory, and an A10@24GB GPU at 2.5 USD/hour).

Following recent studies [29, 34, 45, 57], Grapin stores vertex data entirely in the GPU. This does not affect scalability, as the number of vertices is typically several orders of magnitude smaller than the number of edges [29]. In Grapin, each vertex consumes an average of 88 bytes to maintain the result, dependency, and index data. A commonly used 24GB GPU can support graphs with up to 272M vertices and approximately 7.8B edges, assuming an average vertex degree of 34.8 (derived from the five public graphs). To further improve scalability, a potential solution is to store both vertex and edge data in CPU memory and schedule them in partitions at runtime, following the fully external processing model [51, 58]. Compared to Grapin, this approach incurs additional overhead due to repeated vertex data loading in each iteration [58]. Optimizing fully external graph processing remains an open challenge on GPU-CPU systems, which we leave for future work.

8 Related Work

Incrementalizing graph algorithms. The basic DM algorithm requires the graph algorithm to satisfy monotonicity [42], limiting its application to algorithms such as BFS, CC, and SSSP. GraphBolt [27] and DZiG [26] adapt DM to support Bulk Synchronous Parallel (BSP) semantics by maintaining dependency across iterations. However, they increase the storage overhead from $O(|V|)$ to $O(L|V|)$ (L is the number of iterations), rendering them impractical for GPU deployment [41]. Ingress [15] and iTurboGraph [21] extend DM to support non-monotonic graph algorithms (e.g., PageRank) by transforming them into a monotonic equivalent with similar computation patterns, while maintaining storage overhead at $O(|V|)$.

Streaming graph processing on GPUs. In-memory GPU streaming graph frameworks [3, 6, 16, 37, 48, 49, 52] typically organize the graph in non-contiguous memory slices to support parallel updates. For instance, GPMA [37] divides each vertex's adjacency list into small chunks and reserves gaps between them. faimGraph [48, 49] and SHGraph [3] store edges in multiple blocks and index them through a linked list or hash table. Although these systems achieve excellent performance, they cannot handle large graphs. EGraph [54] introduces an out-of-memory framework that shares commonly accessed subgraphs among multiple concurrent tasks. However, it requires complete data transfer for each task. Grapin-ZC can be viewed as a communication-optimized version of EGraph.

9 Conclusion

We present Grapin, a high-performance out-of-memory GPU streaming graph processing system that minimizes graph data accesses. Grapin achieves its efficiency through two key components for eliminating redundant accesses: 1) an advanced GPU incremental graph computation engine, which addresses the challenge of atomic updates to the result and dependency on GPUs by transforming them into a sequence of GPU-friendly CAS operations; and 2) a GPU dynamic graph management framework that minimizes CPU–GPU data transfers during long-duration computations, using fine-grained and low-overhead graph data caching. Additionally, it improves dynamic graph access performance with layout optimizations. Experimental results show that Grapin can process graphs with billions of edges on a single GPU by effectively reducing data transfers, delivering speedups ranging from 1.8x to 96.9x compared to CPU-based systems.

References

[1] uk-2005, 2005. *https://www.cise.ufl.edu/research/sparse/matrices/LAW/uk-2005.html*.

[2] Tyler N. Allen and Rong Ge. In-depth analyses of unified virtual memory system for GPU accelerated computing. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 64. ACM, 2021.

[3] Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. Dynamic graphs on the GPU. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*, pages 739–748. IEEE, 2020.

[4] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In Vivek Sarkar and Lawrence Rauchwerger, editors, *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 235–248. ACM, 2017.

[5] Nils Boeschen and Carsten Binnig. Gacco - A gpu-accelerated OLTP DBMS. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1003–1016. ACM, 2022.

[6] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–7. IEEE, 2018.

[7] Jing Chen, Qiange Wang, Yu Gu, Chuanwen Li, and Ge Yu. Unified-memory-based hybrid processing for partition-oriented subgraph matching on GPU. *World Wide Web*, 25(3):1377–1402, 2022.

[8] Nvidia thrust, 2024. https://developer.nvidia.com/thrust.

[9] Deep graph library:towards efficient and scalable deep learning on graphs, 2020. https://www.dgl.ai/.

[10] Alibaba cloud services, 2023. https://www.alibabacloud.com/.

[11] Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. Incrementalizing graph algorithms. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 459–471. ACM, 2021.

[12] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 513–527. ACM, 2021.

[13] Per Fuchs, Jana Giceva, and Domagoj Margan. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.*, 15(6):1173–1186, 2022.

[14] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David A. Bader. Traversing large graphs on gpus with unified memory. *Proc. VLDB Endow.*, 13(7):1119–1133, 2020.

[15] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. Automating incremental graph processing with flexible memoization. *Proc. VLDB Endow.*, 14(9):1613–1625, 2021.

[16] Oded Green and David A. Bader. custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–6. IEEE, 2016.

[17] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. VCSR: mutable CSR graph format using vertex-centric packed memory array. In *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022*, pages 71–80. IEEE, 2022.

[18] Cuda threads and atomicsc, 2023. https://mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf.

[19] Xiaolin Jiang, Mahbod Afarin, Zhijia Zhao, Nael B. Abu-Ghazaleh, and Rajiv Gupta. Core graph: Exploiting edge centrality to speedup the evaluation of iterative graph queries. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*, pages 18–32. ACM, 2024.

[20] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[21] Seongyun Ko, Taesung Lee, Kijae Hong, Wonseok Lee, In Seo, Jiwon Seo, and Wook-Shin Han. iturbograph: Scaling and automating incremental graph analytics. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 977–990. ACM, 2021.

[22] Jérôme Kunegis. KONECT: the koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350, 2013.

[23] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 591–600, 2010.

[24] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeon-Gon Cho, and Soojung Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 260–271. IEEE Computer Society, 2014.

[25] Shiyang Li, Ruiqi Tang, Jingyu Zhu, Ziyi Zhao, Xiaoli Gong, Wenwen Wang, Jin Zhang, and Pen-Chung Yew. Liberator: A data reuse framework for out-of-memory graph computing on gpus. *IEEE Trans. Parallel Distributed Syst.*, 34(6):1954–1967, 2023.

[26] Mugilan Mariappan, Joanna Che, and Keval Vora. Dzig: sparsity-aware incremental processing of streaming graphs. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 83–98. ACM, 2021.

[27] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 25:1–25:16. ACM, 2019.

[28] Junyi Mei, Shixuan Sun, Chao Li, Cheng Xu, Cheng Chen, Yibo Liu, Jing Wang, Cheng Zhao, Xiaofeng Hou, Minyi Guo, Bingsheng He, and Xiaoliang Cong. Flowwalker: A memory-efficient and high-performance gpu-based dynamic graph random walk framework. *Proc. VLDB Endow.*, 17(8):1788–1801, 2024.

[29] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. EMOGI: efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 14(2):114–127, 2020.

[30] Seungwon Min, Kun Wu, Sitao Huang, Mert Hidayetoglu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei W. Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endow.*, 14(11):2087–2100, 2021.

[31] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sánchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 1–14. IEEE Computer Society, 2018.

[32] Nsight systems, 2023.

[33] Open computing language opencl, 2023. https://developer.nvidia.com/opencl.

[34] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: minimizing data transfer during out-of-gpu-memory graph processing. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 12:1–12:16. ACM, 2020.

[35] Nikolay Sakharnykh. Everything you need to know about unified memory. https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf, 2018.

[36] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey S. Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, volume 9833 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2016.

[37] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proc. VLDB Endow.*, 11(1):107–120, 2017.

[38] Ruiqi Tang, Ziyi Zhao, Kailun Wang, Xiaoli Gong, Jin Zhang, Wenwen Wang, and Pen-Chung Yew. Ascetic: Enhancing cross-iterations data efficiency in out-of-memory graph processing on gpus. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, pages 41:1–41:10. ACM, 2021.

[39] Thrust, 2023. https://nvidia.github.io/cccl/thrust/api.

[40] Nvidia unified addressing, 2023. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__UNIFIED.html.

[41] Pourya Vaziri and Keval Vora. Controlling memory footprint of stateful streaming graph processing. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 269–283. USENIX Association, 2021.

[42] Keval Vora, Rajiv Gupta, and Guoqing Xu. 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 2017, Xi'an, China, April 8-12, 2017*, pages 237–251. ACM, 2017.

[43] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 38–52, 2019.

[44] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. Grus: Toward unified-memory-efficient high-performance graph processing on GPU. *ACM Trans. Archit. Code Optim.*, 18(2):22:1–22:25, 2021.

[45] Qiange Wang, Xin Ai, Yanfeng Zhang, Jing Chen, and Ge Yu. Hytgraph: Gpu-accelerated graph processing with hybrid transfer management. In *39th IEEE*

International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023, pages 558–571. IEEE, 2023.

[46] Qiange Wang, Yanfeng Zhang, Hao Wang, Liang Geng, Rubao Lee, Xiaodong Zhang, and Ge Yu. Automating incremental and asynchronous evaluation for recursive aggregate data processing. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2439–2454. ACM, 2020.

[47] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: a high-performance graph processing library on the GPU. In Rafael Asenjo and Tim Harris, editors, *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pages 239–252. ACM, 2016.

[48] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 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 2018, Dallas, TX, USA, November 11-16, 2018*, pages 60:1–60:13. IEEE / ACM, 2018.

[49] Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*, pages 1–7. IEEE, 2017.

[50] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.*, 42(1):181–213, 2015.

[51] Tsun-Yu Yang, Cale England, Yi Li, Bingzhe Li, and Ming-Chang Yang. Grafu: Unleashing the full potential of future value computation for out-of-core synchronous graph processing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 467–481. ACM, 2024.

[52] Fan Zhang, Lei Zou, and Yanpeng Yu. LPMA - an efficient data structure for dynamic graph on gpus. In Wenjie Zhang, Lei Zou, Zakaria Maamar, and Lu Chen, editors, *Web Information Systems Engineering - WISE 2021 - 22nd International Conference on Web Information Systems Engineering, WISE 2021, Melbourne, VIC, Australia, October 26-29, 2021, Proceedings, Part I*, volume 13080 of *Lecture Notes in Computer Science*, pages 469–484. Springer, 2021.

[53] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel Distributed Syst.*, 25(8):2091–2100, 2014.

[54] Yu Zhang, Yuxuan Liang, Jin Zhao, Fubing Mao, Lin Gu, Xiaofei Liao, Hai Jin, Haikun Liu, Song Guo, Yangqing Zeng, Hang Hu, Chen Li, Ji Zhang, and Biao Wang. Egraph: Efficient concurrent gpu-based dynamic graph processing. *IEEE Trans. Knowl. Data Eng.*, 35(6):5823–5836, 2023.

[55] Yu Zhang, Da Peng, Xiaofei Liao, Hai Jin, Haikun Liu, Lin Gu, and Bingsheng He. Largegraph: An efficient dependency-aware gpu-accelerated large-scale graph processing. *ACM Trans. Archit. Code Optim.*, 18(4):58:1–58:24, 2021.

[56] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. Graphm: an efficient storage system for high throughput of concurrent graph processing. In Michela Taufer, Pavan Balaji, and Antonio J. Peña, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, pages 3:1–3:14. ACM, 2019.

[57] Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, Hai Jin, Jingling Xue, Zhiyuan Shao, and Qiang-Sheng Hua. Scaph: Scalable gpu-accelerated graph processing with value-driven differential scheduling. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 573–588. USENIX Association, 2020.

[58] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 375–386. USENIX Association, 2015.