TempGraph: An Efficient Chain-driven Temporal Graph Computing Framework on the GPU

Jin Zhao*
Huazhong University of Science and Technology
Wuhan, China
zjin@hust.edu.cn

Qian Wang
Huazhong University of Science and Technology
Wuhan, China
qianwang77@hust.edu.cn

Ligang He
University of Warwick
Coventry, United Kingdom
ligang.he@warwick.ac.uk

Yu Zhang*
Huazhong University of Science and Technology
Wuhan, China
zhyu@hust.edu.cn

Sheng Di
Argonne National Laboratory
Lemont, IL, USA
sdi@anl.gov

Bingsheng He
National University of Singapore
Singapore
hebs@comp.nus.edu.sg

Xinlei Wang*
Huazhong University of Science and Technology
Wuhan, China
xinleiwang@hust.edu.cn

Hui Yu*
Huazhong University of Science and Technology
Wuhan, China
huiy@hust.edu.cn

Hao Qi*
Huazhong University of Science and Technology
Wuhan, China
theqihao@hust.edu.cn

Longlong Lin
Southwest University
Chongqing, China
longlonglin@swu.edu.cn

Linchen Yu
Huazhong University of Science and Technology
Wuhan, China
linchenyu@hust.edu.cn

Xiaofei Liao*
Huazhong University of Science and Technology
Wuhan, China
xfliao@hust.edu.cn

Hai Jin*
Huazhong University of Science and Technology
Wuhan, China
hjin@hust.edu.cn

Abstract

Tackling temporal path problems in temporal graphs is essential for time-sensitive applications. Although many solutions have been proposed to handle temporal path problems, due to the intrinsic time constraints, these solutions require the vertices of the temporal graph to be sequentially handled along the time-dependent chains (i.e., the temporal dependencies between these vertices) to form the temporal path. This sequential temporal nature poses the challenges of *poor parallelism* and *slow convergence speed*, preventing existing solutions from fully leveraging the massive parallelism and high internal bandwidth of GPU to handle temporal path problems. To overcome these challenges, this paper proposes *TempGraph*, an efficient chain-driven GPU-based temporal graph computing framework. Specifically, it transforms the temporal graph into a set of disjoint time-dependent chains that can elegantly expose the temporal dependency between the vertices while facilitating the fast path exploration along these chains over GPU. Furthermore, TempGraph employs a novel *Generate-Activate-Compute* execution model to decouple the temporal dependency between different chains through maintaining a set of shortcuts for them, which enables multiple chains to be concurrently handled by massive GPU threads, achieving fast convergence speed and high

* National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.

ASPLOS '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1080-3/2025/03
https://doi.org/10.1145/3676642.3736116

parallelism on the GPU. Experiments on an A100 GPU show that TempGraph outperforms the state-of-the-art GPU-based solutions by 3.0-16.2×. Besides, TempGraph on an A100 GPU gains 33.9-368.9× speedups compared to the cutting-edge CPU-based system TeGraph on a 128-core CPU machine.

CCS Concepts: **• Computing methodologies → Parallel computing methodologies**; **• Computer systems organization → Parallel architectures**.

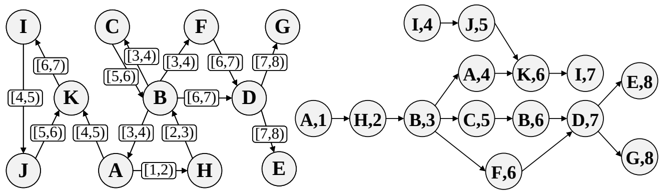
Keywords: Temporal graph computing; GPU; Data parallelism; Convergence speed

ACM Reference Format:
Jin Zhao, Qian Wang, Ligang He, Yu Zhang, Sheng Di, Bingsheng He, Xinlei Wang, Hui Yu, Hao Qi, Longlong Lin, Linchen Yu, Xiaofei Liao, and Hai Jin. 2025. TempGraph: An Efficient Chain-driven Temporal Graph Computing Framework on the GPU. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676642.3736116

1 Introduction

Many real-world graphs are temporal in nature, meaning their edges are annotated with temporal information [28, 46], and such graphs are referred to as *temporal graphs*. This temporal information serves as a crucial metric for data analytics in many domains such as transportation networks [7], social media [45], real-time epidemiology analysis [82], and e-commerce [29]. Figure 1 (a) depicts a temporal graph for an aviation network, where the time interval [4, 5) of the edge $I \longrightarrow J$ signifies the occurrence of a flight departing from I at time 4 and arriving at J at time 5. The *temporal path problems* in the temporal graphs constitute the foundational components for numerous time-sensitive applications. For instance, the temporal path problems is crucial for optimizing routes in traffic navigation [56], tracking the spread of information in social network [63], modeling disease transmission in health informatics [40], and detecting anomalies in financial networks [38]. A temporal path is a legal path under temporal constraints [18–20, 56], where the time sequence along this path is strictly increasing. For example, to find a legal path in an aviation network, the arrival time must be earlier than the departure time at each transit airport.

The wide applicability and criticality of application domains necessitate high performance for tackling temporal path problems. Due to their massive data parallelism, GPUs have become the most widely adopted general-purpose accelerator [11] and thus are attractive for accelerating temporal graph computing (as discussed in §2.2). Recently, many temporal graph computing engines [18, 19, 56, 68–70], primarily designed for CPU platforms, have emerged, typically following two execution models. The first model adapts static graph algorithms, e.g., Bellman-Ford [2] or Dijkstra's algorithm [8],



(a) Temporal Graph (b) Transformed DAG

Figure 1. An example of aviation network

by additionally considering time constraints. This *static execution* model can be implemented using existing GPU-based static graph processing systems [15, 27, 35, 37, 53, 54, 65, 66, 81]. However, it suffers from significant redundant data access and computational overhead due to the costly additional operations required to guarantee time constraints [1, 18, 19]. To address these problems, a *transformation-based execution* model [18, 19, 21, 68, 70] has been proposed. It transforms the temporal graph into a *Directed Acyclic Graph* (DAG) by embedding timing information into the vertices (as illustrate in Figure 1(b)). By this way, the time constraints are naturally reflected by the topological structure of the transformed DAG, which enables to guarantee time constraints with much lower extra overhead. As a result, it demonstrates better performance than *static execution* model [18, 19]. However, naively implementing this advanced CPU-oriented execution model on GPU to accelerate temporal graph computing is still inefficient (§2.3), because the inherent time constraints of temporal path problems may easily make it underutilize the computing power and memory bandwidth of the GPU.

The resource underutilization primarily arises from the sequential nature of vertex state propagation along the time-dependent chains (i.e., the temporal dependencies between the vertices of the legal path) in the temporal graphs. Specifically, each vertex is allowed to propagate its state to its direct neighbor (i.e., this neighbor can be handled) only if its timestamp is earlier than that of this neighbor. This unique characteristic gives rise to two fundamental challenges. First, the inherent time constraints restrict the state propagation between the vertices of the temporal graphs, resulting in only a very small fraction of vertices (less than 6.2% in our characterization) being active in each iteration. Such *poor parallelism* leads to a large number of GPU threads remaining idle. Second, the long time-dependent chains mean that each vertex requires many iterations to propagate its state to its indirect neighbors to trigger the processing atop them, leading to *slow convergence speed*. Consequently, these challenges cause the massive parallelism and high internal bandwidth of GPU to be considerably underutilized, incurring a low GPU utilization (less than 26.4% based on our characterization).

To address the above challenges, we propose a GPU-based temporal graph computing framework *TempGraph* that can efficiently handle the temporal path problems by fully exploiting the parallelism potential of GPU. Specifically, TempGraph transforms the temporal graph into a set of disjoint time-dependent chains that can elegantly expose the temporal order of edges, offering an opportunity for facilitating the fast vertex state propagation along these chains over GPU, where each time-dependent chain is treated as the basic parallel processing unit. Moreover, TempGraph features a novel chain-driven *Generate-Activate-Compute* (GAC) execution model to decouple the temporal dependency among different chains by generating a shortcut between the head vertex and tail vertex of each chain. The shortcuts corresponding to different chains can be generated using massive GPU threads in parallel. Through these generated shortcuts, the head vertex of each chain can directly propagate its state to the tail vertex of this chain. Then, the other chains originating from this tail vertex can be immediately activated and computed without waiting for slow state propagation along the temporal order sequentially. As such, each vertex can propagate its state to other vertices via much fewer iterations and massive chains of the temporal graph can be handled by GPU threads concurrently, ensuring fast convergence speed and high data parallelism when handling temporal path problems. Besides, TempGraph employs a temporal-dependency-aware partition scheduling method along with asynchronous CPU-GPU data transfer to efficiently support out-of-GPU-memory processing of large-scale temporal graphs.

We conduct extensive experiment on both real-world and synthetic datasets. The results demonstrate that TempGraph on an NVIDIA A100 GPU achieves 33.9-368.9× speedups compared to the cutting-edge CPU-based solution, i.e., TeGraph [18, 19], on a 128-core Intel CPU machine. Besides, TempGraph outperforms the cutting-edge GPU-based solutions (i.e., the state-of-the-art GPU-based static graph processing systems Tigr [53], Gunrock [66], LargeGraph [77], and HyTGraph [65] that incorporate the cutting-edge temporal graph computing techniques [18, 19, 56]) by 3.0-16.2× on an NVIDIA A100 GPU.

2 Background and Motivation

2.1 Temporal Graph Computing

Temporal Graph. Different from static graphs, each edge in *temporal graphs* has a *lifespan* with a starting time and an ending time, which indicate the corresponding time interval of existence for this edge. Formally, a temporal graph can be represented by G=(V, E), where V denotes the set of vertices and E is the set of edges. For each edge e in E, we define it as e=(u, v, t, t') (or e=(u, v, t, t', w) for weighted edge), where u, $v \in V$ and there exists an edge from u to v starting at time t and ending at time t'. Furthermore, a path P=$\{e_1, e_2,..., e_n\}$ in the temporal graph is called a legal path when P meets the condition: e_i=(u_i, v_i, t_i, t'_i), e_j=(u_j, v_j, t_j, t'_j), if i<j, then $t'_i \le t_j$. For example, in Figure 1(a), a legal path can move from the edge (C, B, 5, 6) to (B, D, 6, 7), but cannot to (B, A, 3, 4).

Temporal Path Problems. Many real-world time-sensitive applications primarily aim to tackle *temporal path problems* on their temporal graphs [45, 68–70, 80]. We list several most representative temporal path problems as follows [26, 68], where each problem involves a single-source query. Given a vertex $u \in V$ and a time interval [$start$, end], the temporal path problem is to find the temporal path P between the time interval (i.e., $start \le start(P)$ and $end(P) \le end$) for each vertex $v \in V$, where P must satisfy the following conditions.

- **Reachability**: Reachability from u to v means that $\mathbf{P}(u, v)$ is not empty, where $\mathbf{P}(u, v) = \{P : P$ is a temporal path from u to $v\}$.
- **Earliest-arrival Path**: $P \in \mathbf{P}(u, v)$ is the earliest-arrival path between u and v if $end(P)$=$min\{end(P')$:$P' \in \mathbf{P}(u, v)\}$.
- **Fastest Path**: $P \in \mathbf{P}(u, v)$ is the fastest path between u and v if $duration(P)$=$min\{duration(P')$:$P' \in \mathbf{P}(u, v)\}$.
- **Shortest Path**: $P \in \mathbf{P}(u, v)$ is the shortest path between u and v if $dist(P) = min\{dist(P') : P' \in \mathbf{P}(u, v)\}$.
- **Top K Nearest Neighbors**: $\forall u \in K$ and $\forall v \in V/K$, $score(u) \le score(v)$, then K is the set of k-nearest neighbors of the vertex x. $score(u)$ is a self-defined measure function, e.g., the shortest path from x to u.

Execution Model of Temporal Graph Computing. Two main execution models are used in the literature [1, 10, 18, 19, 68, 70] to tackle temporal path problems.

Static Execution. It directly employs traditional static graph algorithms, such as Dijkstra's [8] or Bellman-Ford [2], applied through existing static graph processing systems [53–55, 66], to address temporal path problems by incorporating time constraints. Nevertheless, it suffers from significant redundant data access and computation cost [1, 18, 19], because the costly extra operations are required to guarantee time constraints. For example, when exploring the legal paths originating from C in Figure 1 (a), the edges (B, A, 3, 4), (B, C, 3, 4), and (B, F, 3, 4) need to be loaded and processed within the static execution model, despite the fact that these edges do not satisfy the timing constraints in this case.

Transformation-based Execution. It expands each vertex of the temporal graph into multiple vertices with the same vertex ID but different timestamps based on temporal information [10, 18, 19]. This expansion facilitates the transformation of the temporal graph into an equivalent *Directed Acyclic Graph* (DAG), where the topological structure of this DAG can directly reflect the timing constraint as shown in Figure 1 (b). We can find that the illegal path from C to A via B is naturally eliminated in Figure 1 (b), which avoids the redundant overheads in the static execution model. Then, the legal temporal paths can be explored through a universal single scan over the transformed DAG [18, 19], i.e., sequentially scanning the vertices of this DAG along their topological order, which incorporates all essential time constraints.

2.2 The Need for GPU Acceleration

Many real-world scenarios are time-sensitive [6, 18, 19, 56, 68–70]. For instance, recommendation systems need to frequently examine the shortest path among the users in a large temporal network extracted from the shopping logs (e.g.,

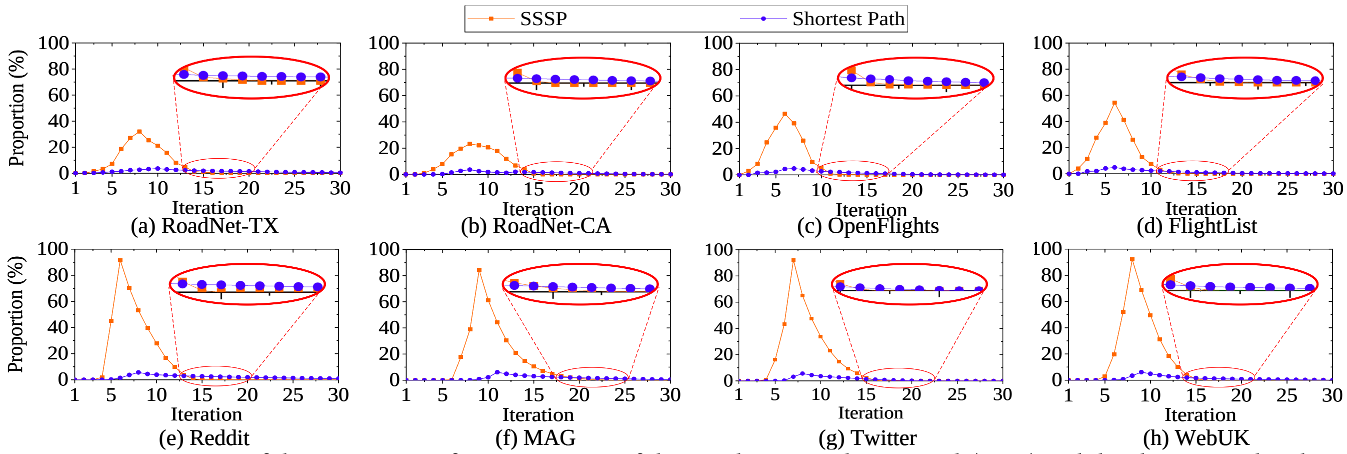


Figure 2. Comparison of the proportion of active vertices of the *Single Source Shortest Path* (SSSP) and the shortest path, where the SSSP disregards the temporal information and the shortest path is executed under time constraints, respectively

more than 1.5 billions sales orders are created in Alibaba's Singles' Day Shopping Festival [33, 64]) with sub-second latency, as part of interactive requests [14, 51, 71]. However, our results reveal that existing CPU-based solutions take over 6 seconds to perform a single shortest path query in a temporal graph with about 1 billion edges, making it challenging to meet time demands. Besides, some applications perform massive numbers of queries on common graphs [18, 56, 73, 79], further exacerbating this challenge. For example, betweenness centrality [4, 47] scenarios launch many independent shortest path queries (each from a random vertex) on the same temporal graph to measure the relative importance of vertices. To address these performance demands, GPUs present a promising alternative to CPU-based solutions due to their higher compute throughput and memory bandwidth.

2.3 Challenges of Efficient Temporal Graph Computing on GPU

During the temporal graph computing procedure, the main operation is to explore the legal paths under time constraints. Although many techniques [18, 19, 68–70, 80] have been designed to enhance the computation and memory efficiency of temporal graph computing, naively porting these CPU-targeted temporal graph computing solutions to GPU is still inefficient due to the following challenges.

Challenge #1 (Poor Parallelism): *The inherent time constraints of the temporal path problems restrict the state propagation between the vertices of temporal graphs, resulting in only a small percentage of vertices being active in each iteration.* Taking Figure 1(a) as an example, the state of vertex *B* cannot be propagated to *A* to trigger the processing of *A* when exploring a temporal path originating from *C*. This is because the ending time of the edge (*C*, *B*, 5, 6) is later than the starting time of the edge (*B*, *A*, 3, 4). In contrast, traditional static graph processing disregards temporal information and allows state propagation from *B* to *A* to trigger the processing of *A*. That is, the average degrees of the real-world temporal graphs will be reduced when the temporal

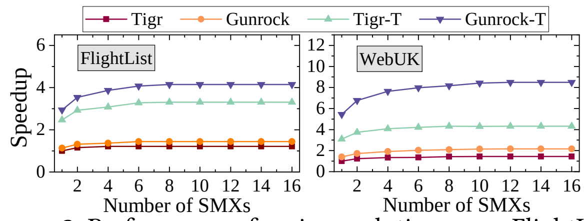


Figure 3. Performance of various solutions over FlightList and WebUK normalized to that of Tigr with one SMX

dimension is considered, which indicates lower achievable parallelism can be achieved when processing each vertex. As a result, compared to static graph processing, the constrained state propagation in temporal graph computing incurs a significantly smaller ratio of active vertices in each iteration. As shown in Figure 2, the ratio of active vertices of temporal graph computing is less than 6.2%, which is much lower than that of traditional static graph processing (up to 92.2%). This indicates that a large number of GPU threads will sit idle when serving temporal graph computing, significantly underutilizing the massive parallelism and high internal bandwidth of GPU.

Challenge #2 (Slow Convergence Speed): *The inherent time constraints of the temporal path problems necessitate that the vertices of the temporal graph be handled sequentially along the temporal dependencies among them, leading to more iterations needed for convergence.* Taking Figure 1(b) as an example and assuming (*B*, 3) is the root vertex for the shortest path, the vertices (*C*, 5), (*B*, 6), and (*D*, 7) can be handled when (*B*, 3) has sequentially propagated its state to them along the time-dependent chain (*B*, 3)⟶(*C*, 5)⟶(*B*, 6)⟶(*D*, 7), which takes three iterations. Consequently, this leads to slow state propagation along the inherent temporal dependencies among the vertices. Moreover, the transformed graphs typically exhibit larger diameters than the original temporal graphs [10, 18, 19], which necessitates more iterations for convergence. Figure 2 shows that the shortest path requires more than 30 iterations over WebUk for convergence, whereas SSSP needs only 16 iterations. Furthermore,

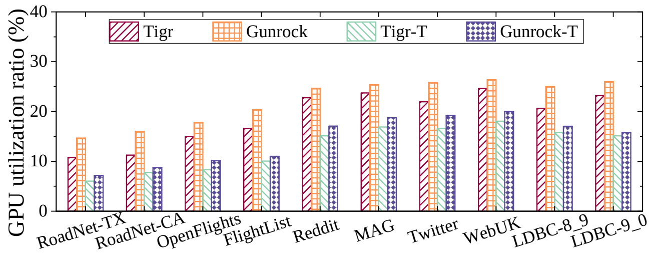


Figure 4. Average GPU utilization ratio of various solutions

the sequential state propagation along the temporal dependencies causes many vertices to remain inactive during execution, exacerbating the challenge of poor parallelism. For instance, only 0.036% of the vertices are active in the 30th iteration of the shortest path over WebUk.

Results. We evaluate four cutting-edge GPU-based graph computing systems, i.e., Gunrock, Tigr, Gunrock-T, and Tigr-T, when running the shortest path over different temporal graphs. Note that Gunrock-T and Tigr-T are the versions of Gunrock [66] and Tigr [53] optimized by the state-of-the-art temporal graph computing solutions [18, 19, 56]. The details of the platform and benchmarks used in this evaluation are introduced in §5.1. Gunrock and Tigr handle the shortest path using the static execution model, while Gunrock-T and Tigr-T apply the transformed-based execution model. Figure 3 evaluates the performance of various solutions running with different numbers of Streaming Multiprocessors (SMXs). Although Gunrock-T performs better than other solutions under all circumstances, its performance improvement still has plateaued as soon as the number of SMXs equals 8 [1] due to the poor parallelism and slow convergence speed. As a result, existing solutions struggle to utilize the massive data parallelism and high internal bandwidth of GPU (e.g., the GPU utilization ratio is less than 26.4% as shown in Figure 4) when handling temporal graph computing.

3 Overview of TempGraph

To address the challenges in §2.3, we propose an efficient GPU-based temporal graph computing framework, called *TempGraph*. It transforms a temporal graph into a set of disjoint time-dependent chains and then efficiently decouples the temporal dependency among these chains via a novel chain-driven *Generate-Activate-Compute* (GAC) execution model. In this way, it enables multiple chains to be efficiently handled by GPU threads in parallel and achieves fast state propagation. This is fundamentally different from existing solutions on GPU, which sequentially handle the edges of the temporal graph along the temporal dependencies among them. In this section, we describe our execution model and the system architecture of TempGraph in detail.

[1]Mainstream GPU accelerators usually have far more than 8 SMXs. For instance, an NVIDIA Tesla P100 has 56 SMXs, while an A100 has been integrated with 128 SMXs.

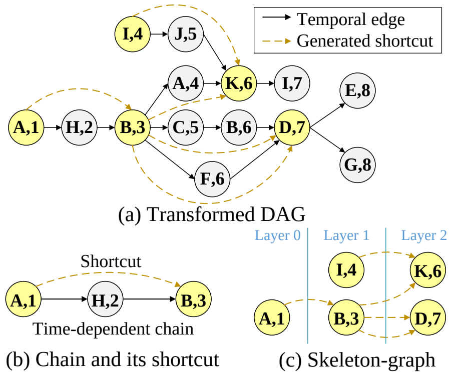


Figure 5. Illustration of chain-driven parallel execution

3.1 Chain-driven Parallel Execution Model

In this subsection, we first present two fundamental concepts and then propose our chain-driven GAC execution model for efficient temporal graph computing on GPU.

Basic Concepts. *Definition 1* (*Time-dependent Chain*): Given a temporal graph G=(V, E), we represent $G = \cup_{Ch_l \in Ch} Ch_l$, where Ch is a set of disjoint time-dependent chains, and $Ch_l = (v_x, t_x) \rightarrow \ldots \rightarrow (v_y, t_y)$ is a sequence of connected vertices that guarantee the time constraints, i.e., $t_x \leq \ldots \leq t_y$. l is the chain *ID*. Moreover, for any two time-dependent chains, e.g., $Ch'_l = (v'_x, t'_x) \rightarrow \ldots \rightarrow (v'_y, t'_y)$ and $Ch''_l = (v''_x, t''_x) \rightarrow \ldots \rightarrow (v''_y, t''_y)$, they should ensure that $Ch'_l \cap Ch''_l \subseteq \{(v'_x, t'_x), (v'_y, t'_y)\} \cap \{(v''_x, t''_x), (v''_y, t''_y)\}$. It means that the intersections of any two chains are only the intersections of the head and tail vertices of these two chains. For faster convergence rate, the disjoint time-dependent chains are the longest chains that satisfy the above conditions. In this way, each time-dependent chain (e.g., $(A, 1) \rightarrow (H, 2) \rightarrow (B, 3)$ in Figure 5(a)) can be efficiently handled by a GPU thread to achieve fast state propagation.

Definition 2 (*Hub-vertex and Skeleton-graph*): We define the set of head vertices of all time-dependent chains as *hub-vertices*, e.g., the vertex $(A, 1)$ in Figure 5(a). Therefore, each *hub-vertex* meets one of the following three conditions: 1) its in-degree is equal to zero; 2) its in-degree is greater than one; 3) its out-degree is greater than one. When we generate a *shortcut* (e.g., $(A, 1) \rightarrow (B, 3)$ in Figure 5(b)) for the head and tail vertices of each time-dependent chain (i.e., $(A, 1) \rightarrow (H, 2) \rightarrow (B, 3)$), the new state of each hub-vertex (e.g., $(A, 1)$) is able to immediately influence another one (i.e., $(B, 3)$) and then quickly drive more chains (i.e., $(B, 3) \rightarrow (A, 4) \rightarrow (K, 6)$, $(B, 3) \rightarrow (C, 5) \rightarrow (B, 6) \rightarrow (D, 7)$, and $(B, 3) \rightarrow (F, 6) \rightarrow (D, 7)$) to be concurrently handled by massive threads of GPU. These *shortcuts* are used to construct the *skeleton-graph*, denoted as G_s, as shown in Figure 5(c).

Chain-driven GAC Parallel Execution. In this execution model, the temporal graph is represented as a series of disjoint time-dependent chains, which are taken as the basic parallel processing unit. The processing of these chains consists of three stages: skeleton-graph generating, shortcut-guided chain activating, and chain-based parallel computing, which are formalized as follows.

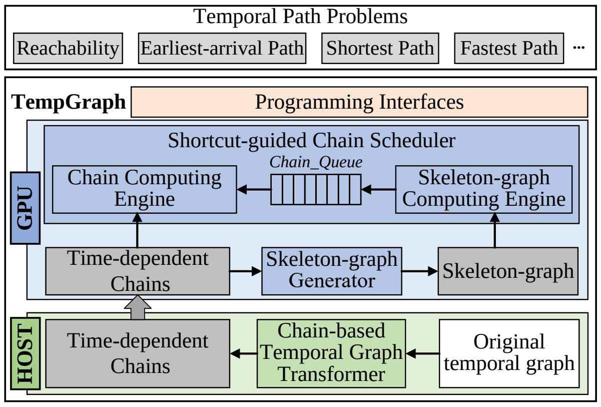


Figure 6. TempGraph architecture

Skeleton-graph Generating. For each time-dependent chain Ch_l=(v_x, t_x)→. . .→(v_y, t_y), it generates a corresponding shortcut S_l: (v_x, t_x)→(v_y, t_y) between Ch_l's head and tail vertices. The weight of each shortcut is calculated according to the specific temporal path problems, because they may use different formulae to calculate the state. For example, the weights corresponding to the shortcut (v_x, t_x)→(v_y, t_y) are $\sum_{e\in Ch_l} W_e$ and $\sum_{e\in Ch_l} duration(e)$ for shortest path and fastest path, respectively, where e is an edge in Ch_l, W_e is the weight of e, and $duration(e)$ is the duration of e, respectively. The generation of the shortcuts for different time-dependent chains can be performed in parallel, because their calculations do not depend on each other. Finally, all constructed shortcuts are used to construct the skeleton-graph G_s=(V_s, E_s), where V_s is the set of hub-vertices and E_s is the set of shortcuts.

Shortcut-guided Chain Activating. To enable different time-dependent chains to be handled in parallel, it immediately propagates the new state of the active vertex to the head vertex of each time-dependent chain (i.e., hub-vertices) using the shortcuts of skeleton-graph G_s. Specifically, it assigns the shortcuts of G_s to be handled according to their topological order and performs the following operation for each shortcut (e.g., S_l: (v_x, t_x)→(v_y, t_y)): $(v_x, t_x).state$ ←`Propagate`(S_l), where $(v, t).state$ is the state of the vertex (v, t) and `Propagate`(∗) denotes the operation that propagates the new state of the vertex (v_x, t_x) based on the shortcut S_l. Note that the `Propagate` operation is determined by the specific temporal path problems. In this way, the head vertices of multiple time-dependent chains will be quickly activated, thereby driving these chains to be concurrently handled by massive threads of GPU.

Chain-based Parallel Computing. When the time-dependent chains are activated, their IDs will be maintained in an active chain queue, i.e., `Chain_Queue`. After that, these activated chains will be allocated to the GPU threads for parallel processing, where each chain is assigned to be handled by a GPU thread. In this way, the GPU can concurrently handle different chains to exploit its high parallelism fully. Furthermore, the new state of each vertex (e.g., $(A, 1)$ in Figure 5) can be immediately propagated to its successors (i.e., $(H, 2)$ and $(B, 3)$ in Figure 5) along the same chain with one iteration.

Table 1. APIs of TempGraph

APIs	Description
`VInitial()`	Initializing the state of each vertex
`GenerateShortcut()`	Generating the shortcut for each chain
`Propagate()`	Propagating the vertex state along each shortcut or each edge of the chain

3.2 System Architecture

Figure 6 shows the architecture of TempGraph, which has the following three main components.

Chain-based Temporal Graph Transformer. The original temporal graph is usually represented in the edge-list format [68, 69]. Before the execution, it first transforms the original temporal graph as a DAG by expanding each vertex based on the timing information [10, 18, 19]. Then, the transformer partitions the DAG into a series of disjoint time-dependent chains. Specifically, it identifies the *hub-vertices* and then concurrently takes these hub-vertices as the roots to explore the DAG so as to construct time-dependent chains. Next, these constructed chains will be transferred into GPU memory for processing. Note that the temporal graph needs to be transformed into the time-dependent chains only once, and then these chains can be reused by different applications.

Skeleton-graph Generator. It generates the shortcuts for time-dependent chains and then uses these generated shortcuts to construct the skeleton-graph. To improve construction efficiency, it assigns multiple GPU threads to calculate the weights of the shortcuts in parallel by concurrently conducting the user-specified operation over multiple chains. Note that the shortcut is generated only if the corresponding chain contains more than two vertices, otherwise the corresponding temporal edge is directly used to construct the skeleton-graph. Then, the constructed skeleton-graph is maintained in GPU memory to guide the parallel computing of the chains. Although generating shortcuts incurs extra runtime overhead, it can effectively leverage massive GPU threads to minimize this cost. More importantly, it enables more chains to be parallelly computed by the GPU during execution (introduced later), significantly boosting the performance of temporal graph computing on GPU.

Shortcut-guided Chain Scheduler. TempGraph employs two separate graph computing engines to handle the skeleton-graph G_s and the time-dependent chains Ch, respectively. During the execution, the skeleton-graph computing engine handles the shortcuts along their topological order in G_s. In this way, the time-dependent chains will be quickly activated for processing. The IDs of these activated chains are then stored in the `Chain_Queue`. Meanwhile, the chain computing engine takes the chains in the `Chain_Queue` for parallel processing, and each activated chain is assigned to be handled by a single GPU thread. By such means, the high parallelism of GPU can be fully exploited to process the temporal graph.

Programming APIs. TempGraph provides several APIs (detailed in Table 1) for users to implement temporal path

```
template<typename T>
// Using the edge ej=<(vj, tj), (vi, ti)> to generate the shortcut
__device__ T GenerateShortcut(Edge ej, Shourtcut Sl){
    return 0
}

// Processing the edge ej=<(vj, tj), (vi, ti)> to propagate the state of (vj, tj) to that of (vi, ti)
__device__ T Propagate(Vertex (vj, tj), Vertex (vi, ti)){
    return min(vj.state, 0)
}
```

(a) Reachability

```
template<typename T>
// Using the edge ej=<(vj, tj), (vi, ti)> to generate the shortcut
__device__ T GenerateShortcut(Edge ej, Shourtcut Sl){
    return ti
}

// Processing the edge ej=<(vj, tj), (vi, ti)> to propagate the state of (vj, tj) to that of (vi, ti)
__device__ T Propagate(Vertex (vj, tj), Vertex (vi, ti)){
    return min(vj.state, ti)
}
```

(b) Earliest-arrival path

```
template<typename T>
// Using the edge ej=<(vj, tj), (vi, ti)> to generate the shortcut
__device__ T GenerateShortcut(Edge ej, Shourtcut Sl){
    return Sl + dist(ej)
}

// Processing the edge ej=<(vj, tj), (vi, ti)> to propagate the state of (vj, tj) to that of (vi, ti)
__device__ T Propagate(Vertex (vj, tj), Vertex (vi, ti)){
    return min(vj.state, vi.state + dist(ej))
}
```

(c) Shortest Path

```
template<typename T>
// Using the edge ej=<(vj, tj), (vi, ti)> to generate the shortcut
__device__ T GenerateShortcut(Edge ej, Shourtcut Sl){
    return Sl + duration(ej)
}

// Processing the edge ej=<(vj, tj), (vi, ti)> to propagate the state of (vj, tj) to that of (vi, ti)
__device__ T Propagate(Vertex (vj, tj), Vertex (vi, ti)){
    return min(vj.state, vi.state + duration(ej))
}
```

(d) Fastest Path

Figure 7. Examples to illustrate the implementation of temporal path problems on TempGraph

problems. `VInitial()` is used to initialize the states of all vertices according to certain applications. `GenerateShortcut()` is employed to generate the weight of the shortcut for each time-dependent chain. For the processing of each shortcut or each edge on a chain, `Propagate()` is used to propagate its source vertex's state to update its destination vertex's state. To illustrate the usage of APIs, we use the shortest path as an example. The `VInitial` function will set the values for the root vertex to 0 and the others to $+\infty$. For each time-dependent chain, the `GenerateShortcut` function is implemented to accumulate the weights of each edge on this chain and use this result as the corresponding shortcut's weight. For the `Propagate` function, we sum the weight of each edge (or shortcut) with the distance of its source vertex to yield a new distance, and then choose the smaller distance between the calculated distance and the destination vertex's distance as its new distance. Our programming APIs only require users to furnish a few lines of code for implementing temporal graph problems as illustrated in Figure 7.

4 Implementation of TempGraph

4.1 Chain-based Temporal Graph Transformation

To provide an opportunity to fully utilize the massive parallelism and high internal bandwidth of GPU, we propose a chain-based temporal graph transformation method. It expands the vertices of the temporal graph to obtain a DAG, liking existing solutions [10, 18, 19], and then partitions this DAG into a set of disjoint time-dependent chains, which can elegantly expose the temporal information with higher data parallelism. The following will mainly describe how to efficiently partition the DAG into time-dependent chains.

Chain-based Temporal Graph Partitioning. We define a vertex $(v, t) \in V$ that meets one of the following conditions as a hub-vertex: 1) $Degree_{\text{in}}(v, t) = 0$; 2) $Degree_{\text{in}}(v, t) \geq 2$; 3) $Degree_{\text{out}}(v, t) \geq 2$, where $Degree_{\text{in}}(v, t)$ and $Degree_{\text{out}}(v, t)$ indicate the in-degree and out-degree of (v, t), respectively. All hub-vertices form a hub-vertex set H. Each of these vertices is usually the head vertex of the time-dependent chains and thus can be used to generate the corresponding chains.

We use a parallel approach to generate time-dependent chains. Specifically, it first divides the set of hub-vertices into several chunks and then assigns them to CPU threads. As illustrated in Algorithm 1, each CPU thread repeatedly takes the hub-vertex as the root and then traverses the transformed DAG in a depth-first order until all assigned hub-vertices have been handled. Note that a hub-vertex (e.g., the vertex $(B, 3)$) may be the head vertex of multiple time-dependent chains (e.g., $(B, 3) \rightarrow (A, 4) \rightarrow (K, 6)$ and $(B, 3) \rightarrow (C, 5) \rightarrow (B, 6) \rightarrow (D, 7)$). For each outgoing direct neighbor (e.g., (v_l, t_l)) of the root, a time-dependent chain (i.e., Ch_l) will be generated by using the ChainPartitioning function (lines 2-5). When all chains originating from this root have been generated, this root will be set as finished (line 6). For the generation of each chain, it first inserts the traversed vertex (e.g., (v_l, t_l)) into a vertex queue of the chain (i.e., Ch_l) and then detects whether this vertex does not belong to the hub-vertices H (lines 9-10). If it does not belong to H, the outgoing direct neighbor of (v_l, t_l) will be further traversed to generate the chain (lines 11-13). The vertices recursively explored by each CPU thread are inserted into the same vertex queue successively to construct chains (lines 3 and 9). When a new chain is generated (lines 14 and 17), it is only necessary to record the offset of the vertex queue to maintain the head vertex of this chain. In this way, the temporal graph can be transformed into a series of disjoint time-dependent chains. Within each chain, the vertices are stored following their inherent temporal order, which facilitates fast vertex state propagation along the chain and enhances the locality of vertex computing.

Algorithm 1 Chain-based Transformation on CPU

```
1:  function TRANSFORMATION(($v_{\text{root}}$, $t_{\text{root}}$), $Ch$)
2:      for each outgoing neighbor ($v_l$, $t_l$) of ($v_{\text{root}}$, $t_{\text{root}}$) do
3:          INSERT(($v_{\text{root}}$, $t_{\text{root}}$), $Ch_l$)
4:          CHAINPARTITIONING(($v_l$, $t_l$), $Ch_l$)
5:      end for
6:      Set the vertex ($v_{\text{root}}$, $t_{\text{root}}$) as finished
7:  end function
8:  function CHAINPARTITIONING(($v_l$, $t_l$), $Ch_l$)
9:      INSERT(($v_l$, $t_l$), $Ch_l$)
10:     if ($v_l$, $t_l$) $\notin H$ then
11:         if ($v_l$, $t_l$) has outgoing neighbor ($v$, $t$) then
12:             CHAINPARTITIONING(($v$, $t$), $Ch_l$)
13:         else
14:             NEWCHAIN($Ch_l$)
15:         end if
16:     else
17:         NEWCHAIN($Ch_l$)
18:     end if
19: end function
```

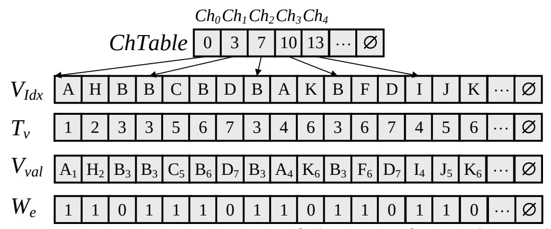

Ch0 Ch1 Ch2 Ch3 Ch4
ChTable 0 3 7 10 13 ··· ∅
V_Idx A H B B C B D B A K B F D I J K ··· ∅
T_v 1 2 3 3 5 6 7 3 4 6 3 6 7 4 5 6 ··· ∅
V_val A1 H2 B3 B3 C5 B6 D7 B3 A4 K6 B3 F6 D7 I4 J5 K6 ··· ∅
W_e 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 0 ··· ∅

Figure 8. Representation of the transformed Graph

Storage of Transformed Graph. We employ five arrays to efficiently store the transformed graph on GPU. V_{Idx} is established to maintain the indexes of the vertices in each chain following their temporal order sequentially. Thus, two successive items in V_{Idx} can represent a temporal edge, where the weight of each temporal edge is stored in W_e. The time information and value of each vertex are stored in T_v and V_{val}, respectively. *ChTable* is also employed to index the chains and maintain the indexes of their head vertices. The range of a chain can be represented by two successive items of *ChTable*. The chains originating from the same vertex will be arranged in consecutive items within the above arrays. This is because these chains are typically activated simultaneously for parallel computing. By such means, we can assign these chains to be concurrently handled by the threads of a warp, enabling these threads to perform coalesced accesses to them (detailed in §4.4). Figure 8 shows the storage of the transformed graph corresponding to the graph in Figure 5(a).

4.2 Skeleton-graph Generation

Although the generated time-dependent chains can achieve fast vertex state propagation along the temporal order implicated in them, the parallel execution of these chains remains challenging due to the intrinsic temporal dependency among them. For example, in Figure 5(a), the computing of the chain Ch_1: $(B, 3)\rightarrow(C, 5)\rightarrow(B, 6)\rightarrow(D, 7)$ depends on that of Ch_0: $(A, 1)\rightarrow(H, 2)\rightarrow(B, 3)$. That is, Ch_1 can be handled only if Ch_0 has been handled and propagated its vertex state to Ch_1. Such poor parallelism makes it struggle to fully utilize the computing power and memory bandwidth of GPU.

To overcome these limitations, we generate the shortcut for each time-dependent chain, which enables us to decouple the temporal dependency between the chains (detailed in §4.3). As depicted in Algorithm 2, in order to generate the shortcuts in parallel, it assigns each chain, e.g., Ch_l, to be handled by a single GPU thread (line 2). After that, it detects whether Ch_l contains more than two vertices (line 3), which can be obtained according to the information in *ChTable*. If so, it repeatedly uses the user-specified function `GenerateShortcut` to calculate the weight of the corresponding shortcut S_l by accumulating the weight of each edge on this chain based on the corresponding application (lines 4-6). Otherwise, the corresponding temporal edge is directly treated as a shortcut to further reduce the extra overhead (line 8). After all the shortcuts are generated, these shortcuts are used to construct the skeleton-graph, which can bring two advantages. First, it enables the vertex state to reach the corresponding indirect neighbors quickly, achieving faster convergence speed for temporal path problems. Second, it enables the chains to be concurrently computed by massive GPU threads through decoupling their temporal dependencies, maximizing the degree of parallelism. Note that when storing the skeleton-graph, the storage order of shortcuts and time-dependent chains is the same. It means that both of them can quickly retrieve each other.

Algorithm 2 Shortcut Generation on GPU

```
1:  function GENERATION($Ch$, $S$)
2:      for each $Ch_l \in Ch$ in parallel do
3:          if $Ch_l$ contains more than two vertices then
4:              for each temporal edge $e \in Ch_l$ do
5:                  $S_l \leftarrow$ GenerateShortcut($S_l$, $e$)
6:              end for
7:          else /*$Ch_l$ only has an edge $e$, i.e., two vertices*/
8:              $S_l \leftarrow W_e$
9:          end if
10:     end for
11: end function
```

4.3 Shortcut-guided Parallel Computing

To efficiently handle the time-dependent chains of the temporal graph, we introduce a novel shortcut-guided hybrid parallel execution method. It embraces two separate graph computing engines (i.e., the skeleton-graph computing engine and the chain computing engine), to decouple the temporal dependency among the chains and handle multiple chains in parallel. However, the implementation of shortcut-guided parallel computing remains the following challenges. First, the vertex states associated with different chains may be irregularly passed through their common neighbor chains, causing redundant data access and computation cost. Second, the skewed lengths of the generated chains may cause the low parallelism of GPU threads. Third, the skeleton-graph struggles to be handled by GPU in parallel, because its shortcuts must be handled along their temporal order. We present how to address these challenges in the following text.

Topology-aware Skeleton-graph Computing. To avoid the redundant computation of chains, the hub vertices of the skeleton graph G_s are assigned to be handled layer by layer along their topological order in G_s (as shown in Figure 5(c)), thereby regularly activating the chains for processing. As depicted in Algorithm 3, it first obtains the corresponding active hub-vertex according to the root vertex v_{root} of the temporal graph application. In detail, it detects whether v_{root} is hub-vertex. If yes, this hub-vertex is set as active. Otherwise, it retrieves the chain (e.g., Ch_x) containing v_{root} (introduced later) and then handles the vertices of Ch_x along their temporal order. Then, the state of the tail vertex of Ch_x will be activated. If this tail vertex is hub-vertex, the

Algorithm 3 Shortcut-guided Hybrid Parallel Execution

```
1:  procedure SkeletonEngine($G_s$, $Ch$, Chain_Queue)
2:      GetActiveHubVertex($Ch$)
3:      for each layer of hub-vertices $H_l$ do
4:          for each active vertex $(v_i, t_i) \in H_l$ in parallel do
5:              for each shortcut $S_l$: $(v_i, t_i) \rightarrow (v_j, t_j)$ in $G_s$ do
6:                  $(v_j, t_j)$.state ←Propagate($S_l$)
7:                  Chain_Queue.Push($Ch_l$.neighborchains)
8:                  Set $(v_j, t_j)$ as active
9:              end for
10:             Set $(v_i, t_i)$ as inactive
11:         end for
12:     end for
13: end procedure
14: procedure ChainEngine($Ch$, Chain_Queue)
15:     RemoveChain($Ch$, Chain_Queue)
16:     SortChain(Chain_Queue)
17:     GroupChain(Chain_Queue)
18:     for each group $gr$ of chains in parallel do
19:         for each chain $Ch_l$ in $gr$ do
20:             ChainComputing($Ch_l$)
21:         end for
22:     end for
23: end procedure
```

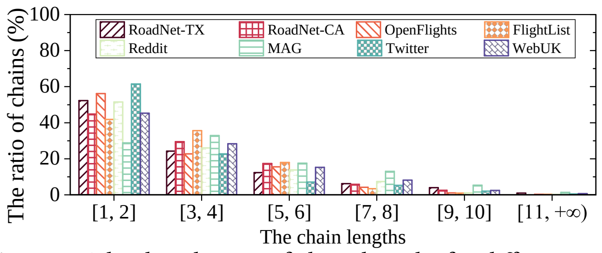


Figure 9. The distribution of chain lengths for different real-world temporal graphs in Table 2

corresponding hub-vertex is set as active. After that, each layer of the hub-vertices is handled in parallel (lines 3-12). For each active hub-vertex (v_i, t_i) in G_s, each of its outgoing edges (e.g., $(v_i, t_i) \rightarrow (v_j, t_j)$) is a shortcut S_l. The processing of S_l can quickly influence the state of another hub-vertex (i.e., (v_j, t_j)) and activate more chains (i.e., the neighbor chains of Ch_l) (lines 5-9), where the activated chains are stored in Chain_Queue (line 7). Note that we establish a hash table on GPU [13] to quickly retrieve the corresponding chain for each vertex (except hub-vertices), where each entry in this hash table is the form of $<(v_i, t_i), Ch_l>$.

Chain-driven Parallel Computing. After the skeleton-graph computing, massive chains will be activated and their IDs are stored in Chain_Queue. Then, these chains are assigned to be concurrently handled by the GPU threads as shown in Algorithm 3. Note that the chains that only contain two vertices do not need to be handled anymore, because their corresponding edges have been handled during the

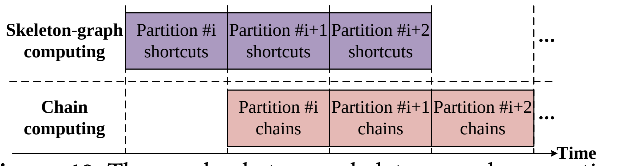


Figure 10. The overlap between skeleton-graph computing and chain computing

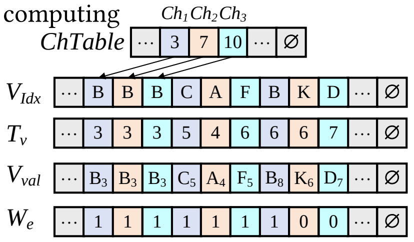


Figure 11. Illustration of the layout optimization

skeleton-graph computing. Therefore, these chains will be removed from Chain_Queue (line 15). For efficient parallel processing, the remaining chains in Chain_Queue will be sorted according to their IDs, ensuring better data locality (line 16). Nevertheless, as illustrated in Figure 9, the generated chains typically exhibit imbalances in their lengths. Due to the lock-step execution semantics of GPU, assigning chains with different lengths to be processed by GPU threads on the same SMX can result in the underutilization of this SMX. Therefore, the chains in Chain_Queue are evenly grouped to try to ensure that different threads on the same SMX handle almost the same number of edges in their assigned chains (line 17). Next, the grouped chains are assigned to be computed by GPU in parallel (lines 18-22).

Overlap between Skeleton-graph Computing and Chain Computing. Although chain computing can be efficiently conducted in parallel, skeleton-graph computing may be the bottleneck due to the lower degree of data parallelism. Fortunately, the interleaving between skeleton-graph computing and chain computing for each partition of shortcuts provides the potential opportunity to hide this bottleneck. Thus, we sequentially divide the shortcuts of the skeleton-graph into a series of partitions according to their topological order, where the shortcuts of the hub vertices in the same layer (as illustrated in Figure 5 (c)) try to be assigned into the same partition. As shown in Figure 10, when the shortcuts of the i^{th} partition have been handled by the skeleton-graph computing engine, the chain computing engine will be driven to handle the chains corresponding to the shortcuts of the i^{th} partition. In the meantime, the skeleton-graph computing engine starts handling the shortcuts of the $(i+1)^{th}$ partition. By such means, the overheads of the skeleton-graph computing can be usually hidden for better performance. Note that these partitions need to be assigned and processed along the topological order among them.

4.4 Layout Optimization for GPU Architectures

Data locality is a pivotal factor influencing the efficiency of GPU applications. During the execution, TempGraph assigns each time-dependent chain to be handled by a thread, where the vertices of this chain are consecutively maintained in V_{Idx} by default as depicted in Figure 8. Therefore, from the perspective of an individual thread, the data accesses to V_{Idx} exhibit good data locality. Nevertheless, GPU threads are organized into warps consisting of 32 threads and operate in a *Single Instruction, Multiple Data* (SIMD) fashion. From the warp's perspective, access to V_{Idx} is actually strided, which hurts the data locality. To overcome this challenge, we optimize the layout of the transformed temporal graph by sorting the items of the chains originating from the same vertex in a strided manner as shown in Figure 11, because these chains are usually activated simultaneously for parallel computing. By such means, when the chains originating from the same vertex are assigned to be handled by the same warp (as they are consecutive in V_{Idx}, T_v, V_{val}, and w_e), each time they access an item, a consecutive segment will be retrieved from global memory, ensuring efficient coalesced accesses.

4.5 Supporting of Out-of-GPU-Memory Processing

To handle large-scale temporal graphs (i.e., those exceeding GPU memory capacity), TempGraph streams the partitions (as introduced in §4.3) from CPU memory to GPU memory for processing. In detail, due to the sequential nature of state propagation along temporal dependencies, the partitions are tried to be dispatched to GPU for parallel processing according to their topological order. Thus, each partition is assigned a layer number, determined by the minimum layer number of its hub vertices. When SMXs become idle, the active partitions with the smallest layer number are prioritized for dispatch from the host to the idle SMXs for processing. To further enhance parallelism, the processing order of the partitions at the same layer is arranged in descending order according to the total number of chains in their successive partitions. By such means, more successive chains are able to be activated and handled when SMXs become idle.

To overlap CPU-GPU data transfer and kernel execution, the partitions are streamed asynchronously to GPU when some previously transferred partitions are being handled on GPU. In detail, TempGraph creates multiple streams for the transfer of partitions, and the number of streams is expected to be N=$(M_G - S_r)/S_P$, where M_G is the GPU global memory size, S_P is the size of each partition, and S_r is the size of the reserved space. Then, the data transfer and kernel computation of different streams will be automatically overlapped.

5 Evaluation

5.1 Experimental Setup

System Configuration. The hardware platform used in our experiments is a server equipped with two 64-core Intel Xeon Platinum 8592V CPUs, 256 GB memory, and an NVIDIA A100 GPU with 128 SMXs (6912 cores) and 80 GB global memory. The server runs Ubuntu 18.04 with Linux kernel version 5.10.0 and CUDA 11.8 installed. All GPU programs are compiled with nvcc using the highest optimization level.

Table 2. Data sets properties ($\overline{L_V}$ and $\overline{L_E}$ are the average lifespans of vertices and edges, respectively)

Graph	$\|V\|$	$\|E\|$	$\overline{L_V}$	$\overline{L_E}$
RoadNet-TX (TX) [32]	1.4 M	2.0 M	19.7	11.2
RoadNet-CA (CA) [32]	2.0 M	2.8 M	20.7	12.4
OpenFlights (OF) [24]	67 K	4.2 M	30.6	24.1
FlightList (FL) [58]	160 K	41 M	56.2	5.6
Reddit (RE) [17]	9.3 M	528.2 M	7.0	1.3
MAG [17]	67.4 M	1.1 B	17.8	12.9
Twitter (TW) [5]	52.5 M	1.9 B	27.5	8.1
WebUK (WU) [3]	133.6 M	5.5 B	16.3	12.5
LDBC-8_9 (L89) [22]	10.6 M	848.7 M	243.5	7.5
LDBC-9_0 (L90) [22]	12.9 M	1.0 B	69.8	44.7

Benchmarks and Datasets. Four representative temporal path problems (i.e., reachability, earliest-arrival path, shortest path, and fastest path, as listed in §2.1) are used as benchmarks. As shown in Table 2, eight real-world graphs and two synthesized graphs are used in our experiments. Note that Reddit, MAG, and WebUK are real-world temporal graphs, while Twitter is a social network with synthesized temporal information [1, 10]. RoadNet-TX and RoadNet-CA are real-world transportation temporal graphs, and OpenFlights and FlightList are real-world flight temporal graphs. LDBC-8_9 and LDBC-9_0 are two synthetic temporal graphs and are generated by using the *Linked Data Benchmark Council* (LDBC) [23, 67].

Baselines. To evaluate the performance of TempGraph, we first incorporate the state-of-the-art temporal graph computing techniques [18, 19, 56] into the cutting-edge GPU-based static graph processing engines, i.e., Tigr (version 1.0) [53] and Gunrock (version 2.1.0[2]) [66], respectively. Then, the optimized versions of Tigr [53] and Gunrock [66] are called Tigr-T and Gunrock-T, respectively, which can handle temporal path problems using the transformation-based execution model. The experiments show that Tigr-T and Gunrock-T outperform Tigr and Gunrock by up to 4.17× and 4.53×, respectively, when handling temporal path problems. Besides, we also use the state-of-the-art CPU-based temporal graph computing engine TeGraph [18, 19] as the CPU baseline. Note that we run the experiments for 100 times with randomly selected vertex as input. The reported results are the average result of the 100 independent runs.

5.2 Overall Performance

Like existing solutions [18, 19, 69, 70], the execution time of TempGraph consists of offline temporal graph transformation time and online temporal graph computation time. In detail, the transformation involves converting the original temporal graph into its equivalent DAG and then partitioning this DAG into a series of disjoint time-dependent chains. The computation includes generating shortcuts for these time-dependent chains and then handling both the shortcuts

[2]Gunrock version 2.1.0 was newly released in July 2024 (https://github.com/gunrock/gunrock/releases/tag/v2.1.0).

Table 3. Transformation time in seconds

	TX	CA	OF	FL	RE
TeGraph	0.044	0.062	0.022	0.135	0.343
Tigr-T	0.051	0.071	0.023	0.144	0.392
Gunrock-T	0.048	0.069	0.024	0.151	0.366
TempGraph	0.053	0.074	0.027	0.169	0.412
	MAG	TW	WU	L89	L90
TeGraph	2.263	2.207	6.407	0.562	0.744
Tigr-T	2.572	2.382	7.014	0.608	0.878
Gunrock-T	2.433	2.314	6.862	0.594	0.841
TempGraph	2.648	2.461	7.376	0.632	0.909

and chains in parallel for certain applications. Note that the offline transformation must be re-performed for each new temporal graph, while the transformed result can be reused for multiple applications running over the same graph. The online computation needs to be performed for each new application. We next report temporal graph transformation time, temporal graph computation time, and end-to-end execution time (i.e., the total time required for both transformation and computation) of different solutions, respectively.

Temporal Graph Transformation. Table 3 shows the transformation cost of different solutions. We can find that TempGraph needs slightly more extra transformation time, accounting for 11.8%-25.2%, 3.1%-17.4%, and 5.3%-12.5% of the transformation time of TeGraph, Tigr-T, and Gunrock-T, respectively. This is because, in addition to transforming the original temporal graph into its equivalent DAG, TempGraph requires a little extra time to divide the transformed DAG into a series of disjoint time-dependent chains by traversing this DAG in parallel for exactly once (detailed in §4.1). Besides, the memory footprint required by TempGraph are 0.5 GB, 0.7 GB, 0.4 GB, 0.9 GB, 4.7 GB, 12.6 GB, 28.4 GB, 49.1 GB, 14.3 GB, and 11.9 GB for TX, CA, OF, FL, RE, MAG, TW, WU, L89, and L90, respectively. The extra storage cost (e.g., maintaining the skeleton-graph and hash table) of TempGraph accounts for 13.8%-27.4% of the size of the original temporal graphs. Although TempGraph needs such additional cost, it offers an opportunity for facilitating the fast vertex state propagation along these chains and enabling many chains to be concurrently handled by massive threads of GPU.

Temporal Graph Computation. Table 4 shows the computation time of Tigr-T, Gunrock-T, and TempGraph on an NVIDIA GPU, and TeGraph on an Intel CPU. It can be seen that TempGraph outperforms TeGraph, Tigr-T, and Gunrock-T by 33.9-368.9×, 5.9-16.2×, and 3.0-11.7×, respectively. The performance improvement of TempGraph mainly comes from the following reasons. First, it enables the vertex states to be propagated more quickly along the time-dependent chains, which elegantly exposes the temporal order of the edge in the temporal graph. Second, it generates the shortcuts to decouple the temporal dependency among different chains, enabling each vertex to quickly propagate its state to its indirect neighbors for faster convergence with fewer iterations. Third, through utilizing the generated shortcuts,

Table 4. Temporal graph computation time in milliseconds

		TeGraph	Tigr-T	Gunrock-T	TempGraph
Reachability	TX	4.9 (49.1×)	0.5 (5.0×)	0.4 (4.1×)	**0.1**
	CA	10.5 (53.5×)	1.1 (5.5×)	0.9 (4.5×)	**0.2**
	OF	4.2 (42.0×)	0.5 (5.0×)	0.3 (3.0×)	**0.1**
	FL	21.1 (70.3×)	2.2 (7.3×)	1.4 (4.7×)	**0.3**
	RE	95.5 (119.4×)	4.7 (5.9×)	3.7 (4.6×)	**0.8**
	MAG	1157.9 (148.5×)	55.2 (7.1×)	40.4 (5.2×)	**7.8**
	TW	1361.5 (172.3×)	64.1 (8.1×)	48.5 (6.1×)	**7.9**
	WU	20667.4 (206.3×)	932.2 (9.3×)	654.6 (6.5×)	**100.2**
	L89	142.8 (129.8×)	6.9 (6.3×)	5.3 (4.8×)	**1.1**
	L90	209.2 (130.1×)	10.5 (6.6×)	7.4 (4.6×)	**1.6**
Earliest-arrival Path	TX	26.2 (65.5×)	3.5 (8.8×)	2.7 (6.8×)	**0.4**
	CA	50.8 (63.5×)	7.1 (8.9×)	4.6 (5.8×)	**0.8**
	OF	28.7 (57.3×)	4.1 (8.2×)	3.2 (6.4×)	**0.5**
	FL	97.4 (64.9×)	13.2 (8.8×)	8.9 (5.9×)	**1.5**
	RE	721.5 (175.9×)	34.1 (8.3×)	24.9 (6.1×)	**4.1**
	MAG	5971.4 (242.7×)	268.5 (10.9×)	191.3 (7.8×)	**24.6**
	TW	6217.8 (244.8×)	284.2 (11.2×)	196.1 (7.7×)	**25.4**
	WU	82750.2 (328.0×)	3672.9 (14.6×)	2404.1 (9.5×)	**252.3**
	L89	718.5 (199.6×)	32.9 (9.1×)	23.3 (6.5×)	**3.6**
	L90	1149.6 (205.3×)	52.3 (9.3×)	36.8 (6.6×)	**5.6**
Shortest Path	TX	34.6 (70.4×)	4.2 (8.4×)	3.8 (7.8×)	**0.5**
	CA	56.4 (64.1×)	6.5 (7.4×)	6.2 (6.9×)	**0.9**
	OF	38.3 (58.2×)	5.4 (8.3×)	6.7 (7.1×)	**0.7**
	FL	101.9 (62.2×)	14.9 (9.1×)	12.1 (7.4×)	**1.6**
	RE	740.2 (189.8×)	34.3 (8.8×)	26.1 (6.7×)	**3.9**
	MAG	6132.1 (249.3×)	278.1 (11.3×)	203.3 (8.3×)	**24.6**
	TW	6295.9 (296.9×)	282.2 (13.3×)	210.2 (9.9×)	**21.2**
	WU	86926.4 (368.9×)	3817.6 (16.2×)	2586.3 (10.9×)	**235.6**
	L89	844.1 (234.5×)	38.3 (10.9×)	28.6 (7.9×)	**3.6**
	L90	1322.9 (249.6×)	59.7 (11.3×)	43.5 (8.2×)	**5.3**
Fastest Path	TX	23.7 (33.9×)	5.4 (7.7×)	4.6 (6.6×)	**0.7**
	CA	44.5 (40.4×)	8.2 (7.5×)	6.9 (6.3×)	**1.1**
	OF	35.2 (44.1×)	6.9 (8.6×)	5.8 (7.3×)	**0.8**
	FL	94.9 (52.7×)	14.1 (7.8×)	12.4 (6.9×)	**1.8**
	RE	718.8 (167.2×)	34.1 (7.9×)	28.4 (6.6×)	**4.3**
	MAG	5959.5 (258.0×)	276.8 (12.0×)	229.1 (9.9×)	**23.1**
	TW	6045.3 (256.2×)	274.4 (11.6×)	239.1 (10.1×)	**23.6**
	WU	84582.7 (350.5×)	3792.9 (15.7×)	2820.3 (11.7×)	**241.3**
	L89	797.2 (194.4×)	37.2 (9.1×)	27.7 (6.8×)	**4.1**
	L90	1246.8 (201.1×)	58.1 (9.4×)	39.0 (6.3×)	**6.2**

massive chains can be activated for computing in a short time, ensuring a higher degree of data parallelism. Note that the CPU-based implementation of our approach only gains 1.5-3.2× speedups than TeGraph [18, 19], which is much lower than the GPU-based implementation.

Figure 12 shows the ratio of the number of active vertices to that of all vertices when running the shortest path. From this figure, we have the following two findings. First, TempGraph requires fewer iterations to converge in comparison with the other solutions. For example, over WebUK, TempGraph converges in only 11 iterations, while Tigr-T and Gunrock-T require more than 30 iterations. Second, the ratio of TempGraph is higher than that of Tigr-T and Gunrock-T, which means that TempGraph can better utilize the massive parallelism and high bandwidth of GPU. Figure 13 shows

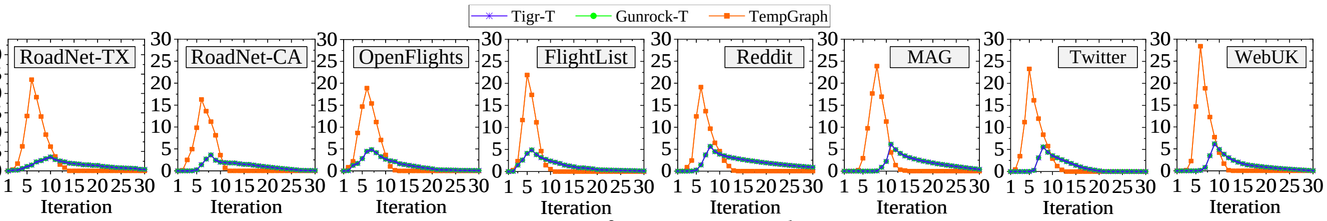


Figure 12. Proportion of active vertices during execution

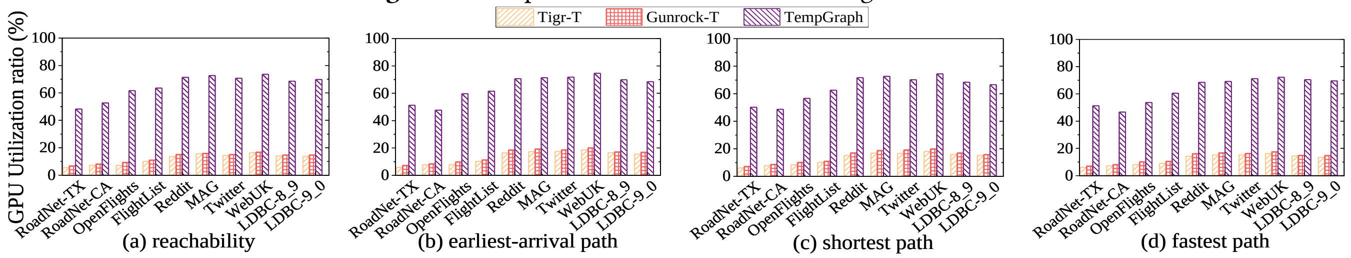


Figure 13. Average GPU utilization ratio of various solutions for different temporal path problems

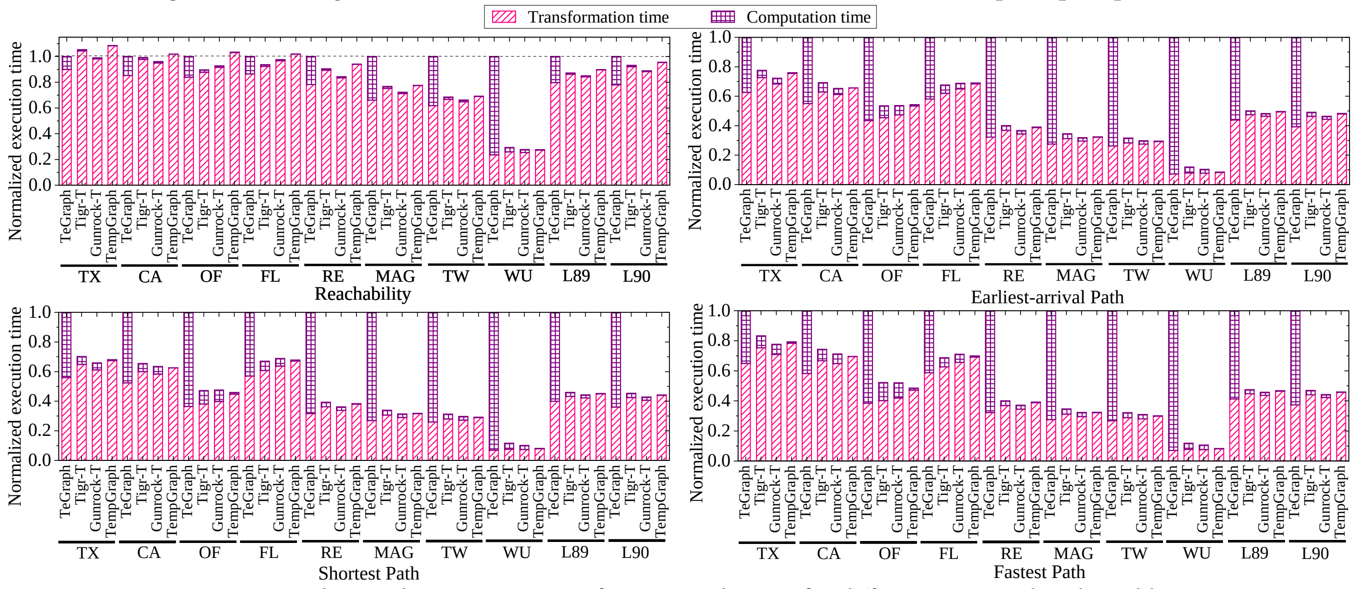


Figure 14. End-to-end execution time of various solutions for different temporal path problems

Table 5. The LDBC datasets properties

Graph	$\|V\|$	$\|E\|$	Graph Size
LDBC-8_9	10.6 M	848.7 M	12.5 GB
LDBC-9_0	12.9 M	1.0 B	14.9 GB
LDBC-9_1	16.1M	1.3 B	19.2 GB
LDBC-9_2	434.9 M	1.0 B	44.7 GB
LDBC-9_3	555.3 M	1.3 B	58.2 GB

that TempGraph (68.4%-74.6%) obtains higher ratios than Tigr-T (13.5%-18.6%) and Gunrock-T (14.7%-20.2%) under all circumstances due to the higher data parallelism and faster convergence speed of TempGraph.

End-to-End Performance. Figure 14 shows that TempGraph achieves higher end-to-end performance than other solutions when handling large-scale temporal graphs (e.g., WebUK). This is because extensive time-dependent chains of these graphs can be concurrently handled by GPU, bringing significant speedups to effectively mitigate the extra transformation cost. In general, the one-time transformation time and the computation time are reported separately [18, 19, 69, 70]. The following results only report the computation time.

5.3 Scalability of TempGraph

Figure 15 evaluates the performance of various solutions over the five synthetic temporal graphs (generated by LDBC [23, 67]) with different graph sizes, where their properties are detailed in Table 5. From Figure 15, we can find that the higher performance improvement is obtained by TempGraph when the size of the graph increases. This is because when handling the larger temporal graphs, TempGraph enables more time-dependent chains of the temporal graph to be handled by GPU threads concurrently, ensuring higher parallelism. This means that TempGraph is more effective for large-scale temporal graph computing, ensuring good scalability.

To further evaluate the scalability of TempGraph, we limited the number of available SMXs during execution over WebUK. As shown in Figure 16, we observe that Tigr-T and

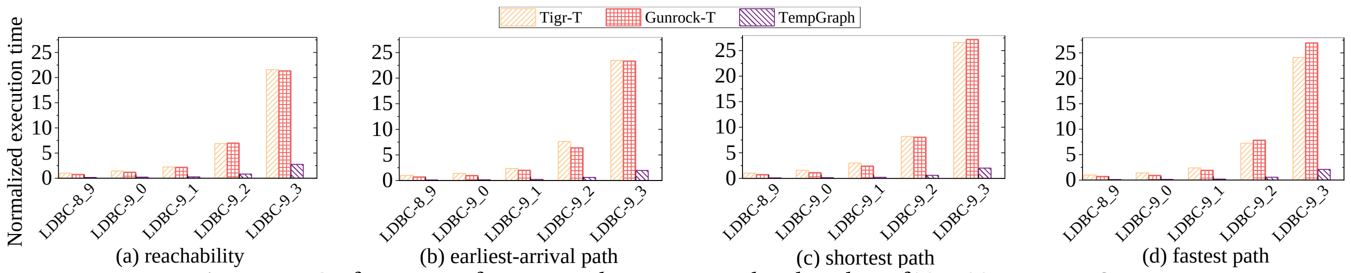


Figure 15. Performance of various solutions normalized to that of Tigr-T over LDBC-8_9

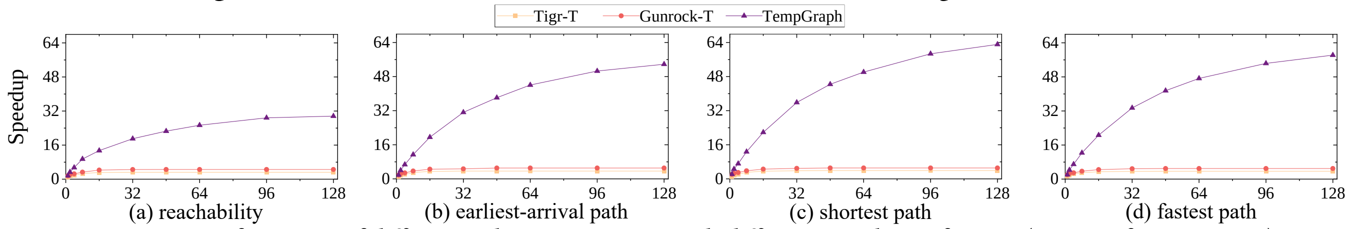


Figure 16. Performance of different solutions running with different numbers of SMXs (ranging from 1 to 128)

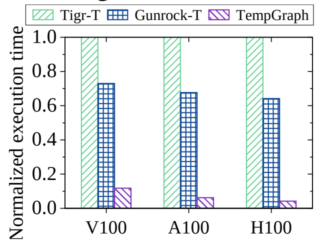


Figure 17. The performance of shortest path over WebUK

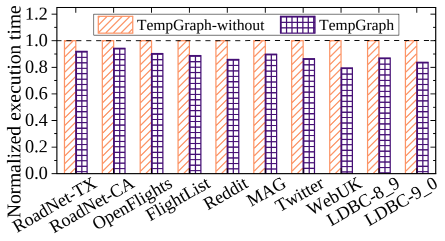


Figure 18. Execution time with/without our GPU-friendly layout optimization

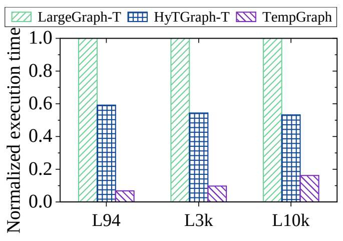


Figure 19. The performance of shortest path

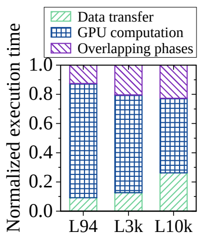


Figure 20. Time breakdown

Gunrock-T experience slow performance improvement as the number of available SMXs increases. This is because that the vertices of the temporal graphs need to be processed along the temporal order sequentially. Therefore, in existing solutions, the vertex states are slowly propagated along the inherent time-dependent chains and only a very small percentage of vertices are active during execution, resulting in a large number of SMXs sitting idle. In contrast, TempGraph can efficiently decouple the temporal dependency among the chains and use the shortcuts to quickly drive massive chains for parallel computing, which boosts a higher degree of data parallelism (which means fewer SMXs sitting idle). As depicted in Figure 16, when the number of available SMXs increases, TempGraph achieves considerable performance improvement until the number of available SMXs equals 96, while this number is around 8 for both Tigr-T and Gunrock-T. Besides, Figure 17 further depicts that TempGraph consistently outperforms other solutions across various GPU types and can achieve greater speedups on more advanced GPUs.

5.4 Performance of Layout Optimization

Figure 18 evaluates the impacts of our layout optimization (detailed in §4.4) on the performance of TempGraph, where TempGraph-without is the version of TempGraph that disenables our layout optimization. The results indicate that our proposed layout optimization yields performance improvements for TempGraph-without, achieving 1.08-1.26× speedups. This is because the enhanced memory locality of our optimization can ensure efficient coalesced accesses, fully exploring the high memory bandwidth of GPU.

Table 6. The large-scale temporal graphs properties

Graph	$\|V\|$	$\|E\|$	Graph Size
LDBC-9_4 (L94)	29.3 M	2.6 B	92.8 GB
LDBC-sf3k (L3k)	33.5 M	2.9 B	114.5 GB
LDBC-sf10k (L10k)	100.2 M	9.4 B	286.2 GB

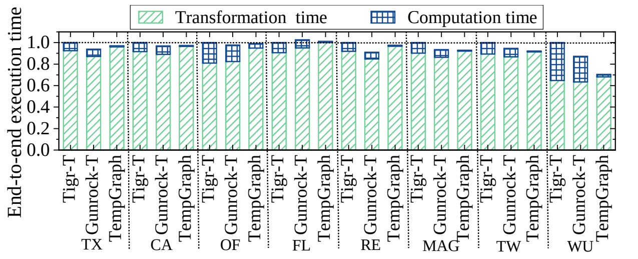


Figure 21. Performance on dynamic temporal graphs

5.5 Performance of Out-of-GPU-Memory Processing

Figure 19 shows the performance of TempGraph compared to that of the cutting-edge out-of-GPU-memory solutions LargeGraph-T and HyTGraph-T when handling large-scale temporal graphs in Table 6. LargeGraph-T and HyTGraph-T are the versions of cutting-edge out-of-GPU-memory graph processing systems LargeGraph [77] and HyTGraph [65] that incorporate [18, 19, 56]. The results show that TempGraph outperforms LargeGraph-T and HyTGraph-T by 6.2-14.7×

and 3.3-8.7×, respectively. Figure 20 further decomposes the total computation time of TempGraph into the time taken by CPU-GPU data transfer, GPU computation, and overlapping phases. The results show that 55.3%–79.8% of CPU-GPU data transfer time can be overlapped in TempGraph.

5.6 Performance on Dynamic Temporal Graphs

Figure 21 evaluates the end-to-end execution time of shortest path over dynamic temporal graphs. Similar to [19], these dynamic graphs are modeled using static graphs with a batch size set to 100 K. This figure shows that TempGraph can achieve higher performance in comparison to other solutions on large-scale dynamic temporal graphs (e.g., WebUK). This is because extensive time-dependent chains of these graphs can be concurrently handled by TempGraph, resulting in significant speedups. This way, the extra re-transformation cost caused by graph updates can be effectively mitigated.

6 Related Work

CPU-based Graph Processing Systems. Over the past decade, numerous CPU-based graph processing systems [12, 34, 36, 42, 43, 49, 50, 59, 72, 78] have been designed. Pregel [39] stands out as one of the earliest distributed systems, using synchronous execution model for graph algorithms. CoRAL [61] and FBSGraph [76] adopt asynchronous execution to ensure fast state propagation and diminish synchronization costs. GraphChi [31] and X-Stream [52] achieve efficient out-of-core graph processing by sequentially accessing storage. To reduce disk I/O cost, Vora et al. [60, 62] propose the dynamic partition and cross-iteration value propagation technique. To reduce repeated graph transfer across the memory hierarchy, Input reduction [30], Wonderland [75], and Core Graph [25] derive a smaller graph for a large graph and employ a two-phase processing method. Pingali et al. [48] try to explore parallelism for irregular applications. However, when applied to temporal graph computing, these solutions incur substantial redundant costs. Besides, the sequential vertex state propagation along temporal dependencies still incurs issues of poor parallelism and slow convergence speed.

GPU-based Graph Processing Systems. The powerful ability of GPU has prompted researchers to propose many GPU-based graph processing systems [15, 27, 35, 37, 81]. Medusa [83] exemplifies the capabilities of GPU-based graph processing. Gunrock [66] performs computation with a data-centric frontier-focused abstraction. LargeGraph [77] uses a path-based approach to handle static graphs on GPU. However, when processing temporal graphs, LargeGraph still suffers from poor parallelism and slow convergence speed due to inherent time dependencies between paths. To alleviate the irregularity of graphs, Tigr [53] proposes a virtual transformation scheme to achieve efficient execution on the GPU. To handle large graphs, Subway [54] dynamically compacts the valid data at runtime to reduce host-GPU communications. To maximize the utilization of host-GPU bandwidth, HyTGraph [65] presents a hybrid data transfer method. However, these systems are mainly designed to process static graphs. When employing them to handle temporal graphs, they suffer from significant redundant data access and computation overhead due to the costly extra operations required to guarantee time constraints.

Temporal Graph Computing Systems. To efficiently handle temporal graphs, many temporal graph computing systems have been proposed recently [9, 41, 68–70]. Chronos [16, 44] uses a locality-aware scheduling method to exploit better data locality. Dynamograph [57] extends Pregel to process large-scale temporal graphs. However, they mainly apply static execution model and gain suboptimal performance due to high time complexity. Thus, some systems use a transformation-based execution model for better performance. ICM [10] extends Pregel to intuitively compose and execute time-dependent graph algorithms, while WICM [1] tries to reduce redundant computations and communications. To further reduce redundant overhead, TeGraph [18, 19] presents a temporal information-aware approach. Srikanth et al. [56] try to accelerate fastest path problem via implementing GPU-based parallel algorithms. Everest [74] focuses on making use of the temporal constraints to generate motif-specific mining code on GPU. However, due to the time constraints of temporal path problems, they still suffer from poor data parallelism and slower convergence speed, struggling to fully utilize the massive parallelism and high internal bandwidth of GPU. In contrast, TempGraph decouples the temporal dependencies among the data and allows these data to be efficiently handled by massive GPU threads in parallel, achieving higher data parallelism and faster convergence speed.

7 Conclusion

This paper proposes a novel GPU-based temporal graph computing framework *TempGraph* to handle temporal path problems efficiently. Specifically, it transforms the temporal graph into a set of disjoint time-dependent chains and decouples the temporal dependency among these chains. By such means, it enables the vertex state to be quickly propagated along these chains and drives these chains to be concurrently handed by massive GPU threads, ensuring fast convergence speed and high parallelism on the GPU. The experimental results show that TempGraph improves the performance by up to 16.2× over the cutting-edge GPU-based solutions.

Acknowledgments

The authors would like to thank our shepherd Rajiv Gupta and all anonymous reviewers for their insightful comments. This paper is supported by National Key Research and Development Program of China (No. 2023YFB4502300) and National Natural Science Foundation of China (No. 62402457 and 62402456). This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357. Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper.

References

[1] Animesh Baranawal and Yogesh Simmhan. 2022. Optimizing the interval-centric distributed computing model for temporal graph algorithms. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 541–558.

[2] Richard Bellman. 1958. On a routing problem. *Quart. Appl. Math.* 16, 1 (1958), 87–90.

[3] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. *SIGIR Forum* 42, 2 (2008), 33–38.

[4] Sebastian Buß, Hendrik Molter, Rolf Niedermeier, and Maciej Rymar. 2020. Algorithmic Aspects of Temporal Betweenness. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2084–2092.

[5] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the Fourth International Conference on Weblogs and Social Media*. 10–17.

[6] Hongtao Chen, Mingxing Zhang, Ke Yang, Kang Chen, Albert Y. Zomaya, Yongwei Wu, and Xuehai Qian. 2023. Achieving Sub-second Pairwise Query over Evolving Graphs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1–15.

[7] Kenneth L. Cooke and Eric Halsey. 1966. The shortest route through a network with time-dependent internodal transit times. *J. Math. Anal. Appl.* 14, 3 (1966), 493–498.

[8] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.

[9] Benjamin Erb, Dominik Meißner, Frank Kargl, Benjamin A. Steer, Felix Cuadrado, Domagoj Margan, and Peter Pietzuch. 2018. Graphtides: a framework for evaluating stream-based graph processing platforms. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems and Network Data Analytics*. 1–10.

[10] Swapnil Gandhi and Yogesh Simmhan. 2020. An interval-centric model for distributed computing over temporal graphs. In *Proceedings of the 36th IEEE International Conference on Data Engineering*. 1129–1140.

[11] Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. ngAP: Non-blocking Large-scale Automata Processing on GPUs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 268–285.

[12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Cmputation on Natural Graphs. In *Proceedings of the 10th USENIX Symposium on Oerating Systems Design and Implementation*. 17–30.

[13] Oded Green. 2021. HashGraph - Scalable Hash Tables Using a Sparse Graph Data Structure. *ACM Transactions on Parallel Computing* 8, 2 (2021), 11:1–11:17.

[14] Qingyu Guo, Fuzhen Zhuang, Chuan Qin, Hengshu Zhu, Xing Xie, Hui Xiong, and Qing He. 2022. A Survey on Knowledge Graph-Based Recommender Systems. *IEEE Transactions on Knowledge and Data Engineering* 34, 8 (2022), 3549–3568.

[15] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 233–245.

[16] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. 1:1–1:14.

[17] Jack Hessel, Chenhao Tan, and Lillian Lee. 2016. Science, AskScience, and BadScience: On the Coexistence of Highly Related Communities. In *Proceedings of the Tenth International Conference on Web and Social Media*. 171–180.

[18] Chengying Huan, Hang Liu, Mengxing Liu, Yongchao Liu, Changhua He, Kang Chen, Jinlei Jiang, Yongwei Wu, and Shuaiwen Leon Song. 2022. TeGraph: A Novel General-Purpose Temporal Graph Computing Engine. In *Proceedings of the 38th IEEE International Conference on Data Engineering*. 578–592.

[19] Chengying Huan, Yongchao Liu, Heng Zhang, Hang Liu, Shiyang Chen, Shuaiwen Leon Song, and Yanjun Wu. 2024. TeGraph+: Scalable Temporal Graph Processing Enabling Flexible Edge Modifications. *IEEE Transactions on Parallel and Distributed Systems* 35, 8 (2024), 1469–1487.

[20] Chengying Huan, Shuaiwen Leon Song, Santosh Pandey, Hang Liu, Yongchao Liu, Baptiste Lepers, Changhua He, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2023. TEA: A General-Purpose Temporal Graph Random Walk Engine. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 182–198.

[21] Silu Huang, Ada Wai-Chee Fu, and Ruifeng Liu. 2015. Minimum Spanning Trees in Temporal Graphs. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 419–430.

[22] Alexandru Iosup, Ahmed Musaafir, Alexandru Uta, Arnau Prat Pérez, Gábor Szárnyas, Hassan Chafi, Ilie Gabriel Tănase, Lifeng Nai, Michael Anderson, Mihai Capotă, et al. 2020. The LDBC Graphalytics Benchmark. *arXiv preprint arXiv:2011.15028* (2020).

[23] Alexandru Iosup, Ahmed Musaafir, Alexandru Uta, Arnau Prat-Pérez, Gábor Szárnyas, Hassan Chafi, Ilie Gabriel Tanase, Lifeng Nai, Michael J. Anderson, Mihai Capota, Narayanan Sundaram, Peter A. Boncz, Siegfried Depner, Stijn Heldens, Thomas Manhardt, Tim Hegeman, Wing Lung Ngai, and Yinglong Xia. 2020. The LDBC Graphalytics Benchmark. *CoRR* abs/2011.15028 (2020).

[24] Junteng Jia and Austin R. Benson. 2018. Detecting Core-Periphery Structure in Spatial Networks. *CoRR* abs/1808.06544 (2018).

[25] Xiaolin Jiang, Mahbod Afarin, Zhijia Zhao, Nael B. Abu-Ghazaleh, and Rajiv Gupta. 2024. Core Graph: Exploiting Edge Centrality to Speedup the Evaluation of Iterative Graph Queries. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 18–32.

[26] David Kempe, Jon M. Kleinberg, and Amit Kumar. 2002. Connectivity and Inference Problems for Temporal Networks. *J. Comput. System Sci.* 64, 4 (2002), 820–842.

[27] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. 239–252.

[28] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2011. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment* 2011, 11 (2011), P11005.

[29] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1269–1278.

[30] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. 2016. Efficient Processing of Large Graphs via Input Reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 245–257.

[31] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.

[32] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[33] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[34] Xiaofei Liao, Jin Zhao, Yu Zhang, Bingsheng He, Ligang He, Hai Jin, and Lin Gu. 2022. A Structure-Aware Storage Optimization for Out-of-Core Concurrent Graph Processing. *IEEE Trans. Comput.* 71, 7 (2022), 1612–1625.

[35] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *Proceedings of the 2019 USENIX Annual Technical Conference.* 411–428.

[36] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[37] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication. In *Proceedings of the 2017 USENIX Annual Technical Conference.* 195–207.

[38] Xiaoxiao Ma, Jia Wu, Shan Xue, Jian Yang, Chuan Zhou, Quan Z. Sheng, Hui Xiong, and Leman Akoglu. 2023. A Comprehensive Survey on Graph Anomaly Detection With Deep Learning. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (2023), 12012–12038.

[39] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data.* 135–146.

[40] Junbin Mao, Hanhe Lin, Xu Tian, Yi Pan, and Jin Liu. 2023. FedGST: Federated Graph Spatio-Temporal Framework for Brain Functional Disease Prediction. In *Proceedings of the 2023 IEEE International Conference on Bioinformatics and Biomedicine.* 1356–1361.

[41] Domagoj Margan and Peter Pietzuch. 2017. Large-scale stream graph processing: doctoral symposium. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems.* 378–381.

[42] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: sparsity-aware incremental processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems.* 83–98.

[43] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth European Conference on Computer Systems.* 25:1–25:16.

[44] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage* 11, 3 (2015), 1–34.

[45] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-Time Dynamic Network Embeddings. In *Proceedings of the 27th World Wide Web Conference.* 969–976.

[46] Raj Kumar Pan and Jari Saramäki. 2011. Path lengths, correlations, and centrality in temporal networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 84, 1 (2011), 016105.

[47] René Pfitzner, Ingo Scholtes, Antonios Garas, Claudio J Tessone, and Frank Schweitzer. 2013. Betweenness preference: Quantifying correlations in the topological dynamics of temporal networks. *Physical Review Letters* 110, 19 (2013), 198701.

[48] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* 12–25.

[49] Hao Qi, Kang Luo, Ligang He, Yu Zhang, Minzhi Cai, Jingxin Dai, Bingsheng He, Hai Jin, Zhan Zhang, Jin Zhao, Hengshan Yue, Hui Yu, and Xiaofei Liao. 2025. OHMiner: An Overlap-centric System for Efficient Hypergraph Pattern Mining. In *Proceedings of the Twentieth European Conference on Computer Systems.* 621–636.

[50] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: A Locality-centric High-performance Streaming Graph Engine. In *Proceedings of the Nineteenth European Conference on Computer Systems.* 33–49.

[51] Shriram Ramesh, Animesh Baranawal, and Yogesh Simmhan. 2021. Granite: A distributed engine for scalable path queries over temporal property graphs. *J. Parallel and Distrib. Comput.* 151 (2021), 94–111.

[52] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* 472–488.

[53] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems.* 622–636.

[54] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems.* 1–16.

[55] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 135–146.

[56] Mithinti Srikanth, Prashant Singh, and G. Ramakrishna. 2024. GPU Algorithms for Fastest Path Problem in Temporal Graphs. In *Proceedings of the 53rd International Conference on Parallel Processing.* 587–596.

[57] Matthias Steinbauer and Gabriele Anderst-Kotsis. 2016. DynamoGraph: extending the Pregel paradigm for large-scale temporal graph processing. *International Journal of Grid and Utility Computing* 7, 2 (2016), 141–151.

[58] Martin Strohmeier, Xavier Olive, Jannis Lübbe, Matthias Schäfer, and Vincent Lenders. 2021. Crowdsourced air traffic data from the OpenSky Network 2019–2020. *Earth System Science Data* 13, 2 (2021), 357–366.

[59] Pourya Vaziri and Keval Vora. 2021. Controlling Memory Footprint of Stateful Streaming Graph Processing. In *Proceedings of the 2021 USENIX Annual Technical Conference.* 269–283.

[60] Keval Vora. 2019. LUMOS: Dependency-Driven Disk-based Graph Processing. In *Proceedings of the 2019 USENIX Annual Technical Conference.* 429–442.

[61] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. 2017. Coral: Confined recovery in distributed asynchronous graph processing. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 223–236.

[62] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *Proceedings of the 2016 USENIX Annual Technical Conference.* 507–522.

[63] Haoran Wang, Licheng Jiao, Fang Liu, Lingling Li, Xu Liu, Deyi Ji, and Weihao Gan. 2023. Learning Social Spatio-Temporal Relation Graph in the Wild and a Video Benchmark. *IEEE Transactions on Neural Networks and Learning Systems* 34, 6 (2023), 2951–2964.

[64] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* 839–848.

[65] Qiange Wang, Xin Ai, Yanfeng Zhang, Jing Chen, and Ge Yu. 2023. HyTGraph: GPU-Accelerated Graph Processing with Hybrid Transfer Management. In *Proceedings of the 39th IEEE International Conference on Data Engineering.* 558–571.

[66] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 1–12.

[67] Jack Waudby, Benjamin A. Steer, Arnau Prat-Pérez, and Gábor Szárnyas. 2020. Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator. In *Proceedings of the 3rd Joint International Workshop on Graph*

Data Management Experiences & Systems and Network Data Analytics. 8:1–8:8.

[68] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path Problems in Temporal Graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.

[69] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. 2016. Efficient Algorithms for Temporal Path Computation. *IEEE Transactions on Knowledge and Data Engineering* 28, 11 (2016), 2927–2942.

[70] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *Proceedings of the 32nd IEEE International Conference on Data Engineering*. 145–156.

[71] Liang Xiang, Quan Yuan, Shiwan Zhao, Li Chen, Xiatian Zhang, Qing Yang, and Jimeng Sun. 2010. Temporal recommendation on graphs via long- and short-term preference fusion. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 723–732.

[72] Chengshuo Xu, Keval Vora, and Rajiv Gupta. 2019. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 587–600.

[73] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2023. Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 78–92.

[74] Yichao Yuan, Haojie Ye, Sanketh Vedula, Wynn Kaza, and Nishil Talati. 2023. Everest: GPU-Accelerated System For Mining Temporal Motifs. *Proceedings of the VLDB Endowment* 17, 2 (2023), 162–174.

[75] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 608–621.

[76] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Bing Bing Zhou. 2018. FBSGraph: Accelerating Asynchronous Graph Processing via Forward and Backward Sweeping. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2018), 895–907.

[77] Yu Zhang, Da Peng, Xiaofei Liao, Hai Jin, Haikun Liu, Lin Gu, and Bingsheng He. 2021. LargeGraph: An Efficient Dependency-Aware GPU-Accelerated Large-Scale Graph Processing. *ACM Transactions on Architecture and Code Optimization* 18, 4 (2021), 58:1–58:24.

[78] Jin Zhao, Yu Zhang, Ligang He, Qikun Li, Xiang Zhang, Xinyu Jiang, Hui Yu, Xiaofei Liao, Hai Jin, Lin Gu, Haikun Liu, Bingsheng He, Ji Zhang, Xianzheng Song, Lin Wang, and Jun Zhou. 2023. GraphTune: An Efficient Dependency-Aware Substrate to Alleviate Irregularity in Concurrent Graph Processing. *ACM Transactions on Architecture and Code Optimization* 20, 3 (2023), 37:1–37:24.

[79] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis*. 3:1–3:14.

[80] Yifeng Zhao, Xiangwei Wang, Hongxia Yang, Le Song, and Jie Tang. 2019. Large Scale Evolving Graphs with Burst Detection. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. 4412–4418.

[81] Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, Hai Jin, Jingling Xue, Zhiyuan Shao, and Qiang-Sheng Hua. 2020. Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven ifferential Scheduling. In *Proceedings of the 2020 USENIX Annual Technical Conference*. 573–588.

[82] Yunling Zheng, Zhijian Li, Jack Xin, and Guofa Zhou. 2021. A Spatial-temporal Graph based Hybrid Infectious Disease Model with Application to COVID-19. In *Proceedings of the 10th International Conference on Pattern Recognition Applications and Methods*. 357–364.

[83] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1543–1552.