C^{2}graph: A Compression-Collaboration Algorithm for CPU-GPU Hybrid Weighted Graph Traversals

Ning Wang[1], Huaibei Li[2], Shen Su[1], Yu Gu[3*], Ge Yu[3], Zhigang Wang[1*], Dawei Zhao[4], Hui Lu[1], Zhihong Tian[1]

[1]*Cyberspace Institute of Advanced Technology, Guangzhou University,*
Guangdong Key Laboratory of Industrial Control System Security, Guangzhou, China
[2]*Faculty of Information Science and Engineering, Ocean University of China, Qingdao, China*
[3]*School of Computer Science and Engineering, Northeastern University, Shenyang, China*
[4]*Key Laboratory of Computing Power Network and Information Security, Ministry of Education,*
and Qilu University of Technology (Shandong Academy of Sciences), Jinan, Shandong
[1]{wangning,sushen,luhui,tianzhihong}@gzhu.edu.cn, wangzhiganglab@gmail.com,
[2]lihuaibei7951@stu.ouc.edu.cn,[3]{guyu,yuge}@mail.neu.edu.cn,[4]zhaodw@sdas.org

***Abstract*—Most graph algorithms need to iteratively traverse vertices along a large number of edges as well as their weights, rendering them inefficient in both time and space complexities, especially when multi-tenants issue many traversal queries on a given graph. Traditional distributed/parallel solutions mitigate these challenges at the expense of huge hardware investments and high communication delay. Recent studies resort to the modern commodity GPU accelerator and possibly CPUs equipped on a single device, but still face limitations on the memory scalability and the underutilization of compute power.**

This paper investigates the redundant traversal behaviors and accordingly presents a compression algorithm to prune not only graph topology but also edge weights, reducing noticeable memory usage. We compensate for the lost messages caused by pruned edges and give a theoretical guarantee for correctness. Further, we capture the dynamic workload variation of a traversal query. Motivated by the peak-compute requirements, multiple traversals can adaptively select reasonable accelerators between GPUs and CPUs. Our query optimization minimizes the CPU-GPU collaboration cost and fully utilizes their compute power, improving significant query throughput. Extensive experiments show that compared against state-of-the-art techniques, our compression technique achieves up to 36% memory reduction in the case of a single traversal query. When processing multiple traversal queries, our new collaboration framework achieves a 1.9X runtime decrease.

***Index Terms*—weighted graph traversal, GPU parallel computing, graph compression, CPU-GPU collaboration.**

I. Introduction

Graph algorithms excel in analyzing complex relationships among entities/vertices, by iteratively traversing weighted edges between any pair of vertices [1], [2]. In the big data era, frequent iterations over billion-edge graphs strain memory and computing capabilities. Moreover, many traversal queries can be issued by multi-tenants with different sources [3], that exacerbates the already strained resources. Traditionally, a distributed system on a cluster with many CPU devices is a preferred solution for prominent scalability in terms of memory and compute power [4], [5]. However, the hardware investments and maintenance costs are prohibitively expensive, that is out of reach for many scientists, especially in academia.

The modern Graphics Processing Units (GPUs) with massive threads have strong computing power, providing another easy-to-build and affordable alternative for fast traversal on weighted graphs. However, there exist two bottlenecks challenging the practical performance. The one is a GPU card with limited memory capacity cannot accommodate the increasingly growing input graph. The other is the underutilization of massive threads in our focused traversal analytics. This is because vertices are gradually involved in computations and communicate with different numbers of neighbors, leading to irregular behaviors against the regularized Single-Instruction-Multiple-Data (SIMD) hardware design.

For memory scalability, some works increase the provided resources by aggregating multiple GPU accelerators and/or accompanied CPUs on a single device. They optimize the collaboration cost by asynchronously transferring data [6], [7], carefully partitioning graph [8]–[10], smartly managing buffer resources [11], [12], and even a hybrid solution [13]. However, the locality of graph topology is poor because edge links can be freely established between any pair of source and target vertices. Traversal algorithms further complicate the data access pattern since vertices become active for updates once they receive messages delivered along these freely-connected edges. As a result, the frequency of collaboration behaviors across GPUs (and possible CPUs), as well as the volume of transferred data, can be still considerable. The two factors respectively interrupt computations on GPUs and exhaust limited bandwidth. Differently, other works attempt to reduce the storage requirement. They either replace common edges distributed in different adjacency lists with a single copy [14], [15], or re-encode target vertices' ids with fewer bits [16], [17]. However, these compression techniques only focus on graph topology. They cannot work for edge weights with distinct values. In particular, the replacing compression adds the single copy as a new list, which increases the graph diameter and

[*] Yu Gu and Zhigang Wang are corresponding authors.

hence delays the convergence of traversal algorithms.

Fully utilizing the power of GPUs is another research focus. Early efforts like Cooperative Thread Array (CTA) [18]–[20] and Memory Coalescing [21] solve the issues of thread divergence and inefficient GPU memory access, caused by the power-law degree distribution [22]. Notably, the workload of a traversal job dynamically changes. The most recent CPU-GPU collaborative work enhances CTA for workload balance [23]. It even migrates the job from GPU to CPU to avoid cost-ineffective parallel computations. However, in the scenario of a single job, this migration renders all GPU threads idle, leading to heavy waste. Some researchers improve the power utilization by concurrently running multi-sources' traversals [24], but the peak memory and compute consumption can easily exceed the hardware limitation. Also, many threads still remain idle when few vertices are active during the low workload stage.

Clearly, there is an imperative need for sound and effective multi-sources' traversal analytics on weighted graphs. This paper explores a new path to pursue this target by time-space efficient compression and query optimization.

Different from the common-edge-replacing compression, our new design aims to detect redundant paths between any pair of source-target vertices. Accordingly, we preserve a single path for normal message propagation while destroying others by pruning selected edges within them. This is independent of the exact value-based comparison, like the repeated appearance of the common edges. It thereby can normally prune both edges and weights. We give principles to select the preserved path and merge weights of pruned edges on other paths, to output correct results in downstream analytics. We give a theoretical analysis about our correctness guarantee specific to algorithms. In particular, we prioritize vertices by their degrees to skip invalid detections where no path can be pruned. Note that our pruning compression is compatible with existing re-encoding studies since the latter work on the representation of target ids, rather than our focused paths.

Our traversal algorithms also run on the easy-to-build CPU-GPU hybrid framework. Since the memory bottleneck can be effectively mitigated by our new compression technique, now we employ CPUs to help improve the power utilization of GPUs, instead of extending the memory capacity. We make high-level query optimization among different traversal queries, instead of fine-grained iterations within a certain query. More specifically, we continuously track the workload variation of a running query. This query is migrated onto CPUs once its parallel computations on GPUs are no longer cost-effective. The next query with another source vertex is then scheduled onto GPUs. In general, in this pipeline manner, for every traversal query, GPUs always run its iterations with high workloads (many vertices are active) while CPUs run others. This high-level optimization naturally avoids frequent collaboration caused by the poor topology locality, and enables integration of existing CTA and memory coalescing techniques. In particular, we replicate the compressed graph onto GPUs and CPUs to reduce the volume of transferred data. We just migrate the computation logic in a lightweight manner. Moreover, we establish a performance estimation model to predict the runtime of given workloads respectively on CPUs and GPUs. We accordingly make a smart migration decision to keep both of the two accelerators busy. In practice, the decision can be fine-tuned based on the real-time utilization, to further boost the overall throughput.

Below, Sec. II overviews related works. Sec. III gives a background introduction. Sec. IV presents our compression algorithm. Sec. V introduces our collaborative framework. Sec. VI reports experiments. Sec. VII concludes this paper.

II. Related Work

A. Distributed/Parallel Graph Processing

Distributed/parallel systems can provide scalable memory capacity and compute power, by continuously adding new devices/accelerators. Representatives include distributed CPU frameworks like Pregel [4], PowerGraph [22], and HGraph [25], and parallel GPU frameworks running on multiple GPU cards on a single device, like Medusa [26], Gunrock [19] and Groute [18]. This solution significantly increases the hardware investments and maintenance costs. Moreover, computations across devices/accelerators inevitably generate expensive communication and synchronization costs, and also complicate the system design. In particular, the low bandwidth of connections across GPUs heavily limits their throughput, rendering the GPU power underutilization.

B. Centralized CPU-GPU Graph Processing

Notably, GPUs cannot work without CPUs and the latter are usually equipped with large main memory. Researchers thereby propose to run graph analytics with CPUs and GPUs on a single device, to facilitate the system design. A straightforward collaborative manner is to divide the large graph into several small subgraphs, and then load the latter into GPU on demand for fast computations. Now the PCIe bandwidth between GPU and CPU becomes the performance bottleneck. Kim et al. overlap GPU computations and data transferring operations to partially hide the runtime delay [6]. Sengupta et al. track active vertices/edges to reduce the volume of physically loaded data [27]. Researchers also attempt to reduce the number of loading subgraphs by improving topology locality, such as reordering vertices by their degrees [8], pinning subgraphs with many active vertices in the GPU memory in a static or a dynamic tracking manners [12], [28]. These techniques have made great advancements but the performance is still far from ideal, due to the naturally poor locality of graph topology and the huge gap between the limited PCIe bandwidth and the strong GPU power.

Recently NVIDIA released their GPU products with Unified Memory (UM) optimization where all memory resources are uniformly used and pages are automatically swapped between CPUs and GPUs [29]. However, the poor locality of graph traversal analytics leads to many page faults, which frequently interrupts computations on GPUs. To alleviate this negative impact, Wang et al. prioritize graph topology and vertices' attributes resident in GPU [11], and Gera et al. reorder vertices by harmonic centrality to improve the cache hit ratio [9].

Different from the static partitioning techniques mentioned above, some efforts are devoted into dynamically generating subgraphs for GPUs. Sabet et al. simply build the subgraph by active information using CPUs, and then transfer it into GPUs, to overlap the generation and the computation [7]. Zhang et al. propose a path-based rule where vertices in the subgraph can be used in several incoming iterations, that reduces the generation frequency [10]. CGgraph explores degree- and BFS-based rules [23]. It generates subgraphs only once and then keeps them resident in GPUs for multiple usages. That minimizes the generation and transferring costs, but incurs additional synchronization costs for messages between GPU-resident and CPU-resident subgraphs, as validated in our experiments. The most recent *HyTGraph* [13] devises a hybrid framework that smartly runs UM or efficient subgraph generation techniques mentioned above, according to runtime workloads.

C. Graph Compression

Graph compression is another path to reduce the storage requirement and possibly the computation complexity. Existing works basically fall into two categories as given below.

There exists natural redundancy for edges across adjacency lists, because many sources can link to the common set of targets. Such common edges can be directly identified and replaced by a single new virtual vertex to reduce the topology size, where the virtual vertex and the common edges form a new adjacency list for correctness [15]. Motivated by the text redundancy detection [30], F. Claude et al. propose a rule-based recursive replacement policy where more fine-grained common set can be found and replaced [31]. The recent work CompressGraph follows this design but goes further [14], [32]. It builds an end-to-end compression-computation framework. It prioritizes different common sets based on their frequency in the graph, achieving an overall success among compression time, compression ratio, and downstream computation runtime (only for full active algorithms like PageRank). However, these replacement techniques cannot compress edge weights due to the distinct values. Besides, newly added virtual lists increase the graph diameter, which forces traversal analytics to make a long detour to converge. As a result, they cannot work well for our focused weighted graph analytics.

Another compression line is to encode the target vertex ids with fewer bits. Early pioneers capture the value-similarity among target ids within an adjacency list. They encode ids with the small difference gaps instead of the large original values, for a memory-efficient representation [16]. Following that, some graph reordering methods such as LLP [33] and BP [34] are proposed to narrow the difference gaps, to further decrease the number of required bits. The graph system Ligra+ employs this technique [35]. Instead of a user-specified bit number, it uses the default basic data type in encoding for fast decompression. The GPU-friendly variants are also designed for memory-efficient computations [17], [36]. We should stress that the encoding compression is complement to other techniques including our proposals. The new variant of CompressGraph has integrated it to boost the compression ratio [32]. We will also investigate it as future works.

Additionally, a number of studies have explored graph indexing techniques to accelerate traversal and potentially reduce memory usage [37]–[39], at the cost of storing the indices themselves. In contrast, our approach directly modifies the underlying graph structure, offering a space-efficient solution that is complementary to existing indexing methods.

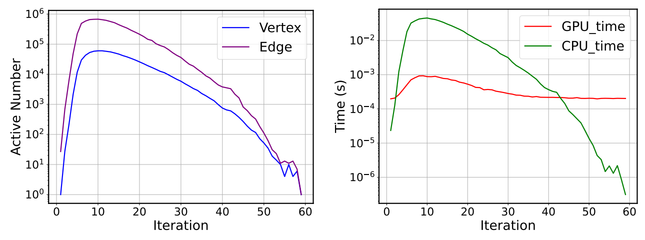
III. Preliminary

A directed weighted graph $G=\{V,E,W\}$ consists of a set of vertices V and a set of edges E. Given a directed edge $e_{ij}\in E$, it links a source vertex $v_i\in V$ and a target vertex $v_j\in V$, with a weight $w_{ij}\in W$. The in-neighbors of v_i, denoted by $\Gamma^{in}(v_i)$, are vertices that have an edge linking to it. Similarly, its out-neighbors $\Gamma^{out}(v_i)$ are vertices that v_i has an edge to link. Its in/out-degree is the number of in/out-neighbors.

Graph algorithms are usually traversal-based. A traversal process begins from one or more given source vertices and then iteratively explores the entire graph along the outgoing edges, typically following a breadth-first search (BFS) approach. This process involves extensive computation and message passing. Each vertex has two states, active or inactive, indicating whether or not it should participate in computations. In each iteration, active vertices first compute received messages, update their vertex values $\mathcal{V}(v_i)$ according to algorithm-specific rules, and then uniformly broadcast messages along outgoing edges. Upon receiving new messages, neighboring vertices determine whether or not they meet the activation conditions by algorithm-specific boundary detection conditions. If the condition is satisfied, these neighboring vertices become active and will be processed in the next iteration.

It is evident that the number of vertices involved in each iteration varies dynamically. Fig. 1(a) shows the variation curve of workloads measured by the number of active vertices, using a well-known Personalized PageRank (PPR) algorithm on the Arabic dataset (see details in Table I). We clearly see the number of active vertices and associated edges quickly increases (divergence) and then gradually decreases (convergence) during iterations. In particular, there is a long convergence with low workloads. Fig. 1(b) shows the variation curve of runtime among iterations on both CPUs and GPUs. We observe that the variation of GPUs is very gentle, compared with CPUs. This is due to the SIMD hardware design and the warm-up cost in parallel computations [40]. Together with Fig. 1(a), we know CPU is more sensitive to the variation of workloads. It is particularly efficient when the load becomes low. But at the divergence stage with high workloads, GPU beats CPU due to the high parallelism. Based on these observations, it is necessary to schedule computations between CPUs and GPUs for fully utilization of compute resource.

Many graph algorithms fall into this traversal category, such as the classic Single-Source-Shortest Path (SSSP) [4], the recommendation algorithm Adsorption used in Youtube [41], Katz metric, and some advanced algorithms like Betweenness



(a) Workload variation (b) Runtime variation

Fig. 1: Features of a traversal algorithm (PPR)

Centrality based on SSSP. In real-world, multi-tenants usually issue a large amount of traversal queries with different source vertices, to perform personalized analysis. The query optimization becomes particularly important in this scenario.

IV. Graph Compression based on Pruning Redundant Paths

Different from existing compression techniques that only work for edges, now we present our new alternative. It identifies redundant paths to guide the pruning of edges as well as weights. Meanwhile, some lightweight compensations are required to guarantee the correctness of downstream analytics.

A. *Overview Design*

During iterations of a traversal job, a vertex v_x is usually updated multiple times, since the received messages arrive at different times/iterations. In fact, the underlying reason is that v_x is located at different message propagation paths. As shown in Fig.2(a), v_4 is an intersection of four paths P_1, P_2, P_3, and P_4. Clearly, v_4 will receive four messages along these paths. The message w.r.t. the shortest path P_4 arrives at the earliest time while others arrive later. Notably, some paths have the same source vertex, like P_1 and P_2 w.r.t. v_3, indicating a strong reachability between v_3 and v_4. In other words, even though some path is destroyed by pruning edges, the algorithm-specific information from the source v_3 can still be delivered to the target v_4 along other paths. Definition 1 mathematically defines this redundant path phenomenon. Inspired by this, we can prune some of edges as well as weights in redundant paths for memory reduction.

Definition 1: Redundant Paths. Two distinct paths P_x and P_y are considered mutually redundant, if they both originate at the same source vertex v_s and terminate at the same target vertex v_t.

Intuitively, among several paths between a pair of source-target vertices, only one needs to be preserved while others are redundant and can be pruned for memory efficient storage. However, a blind pruning operation might generate information loss for which we cannot compensate. Below we give our **triangle-based pruning** framework with a series of principles, to identify edges that can be safely pruned. We repeat these procedures for every pair of source-target vertices, to complete the compression task.

- Enumerating all paths R_{st} for a pair of given source-target vertices (v_s,v_t).
- Identifying the preserved path $\hat{P}_{st} \in R_{st}$ that directly links source and target vertices. Because no intra-vertex exists in $\hat{P}_{st}$, the weight w_{st} w.r.t. the single edge e_{st} can be safely modified for correctness compensation, without impact on other vertices (see details in Sec. IV-B).
- Refining ($R_{st} - \{\hat{P}_{st}\}$) as candidate pruning subset $\bar{R}_{st}$ by excluding any path P_i, if there exist two or more intra-vertices in P_y or the degree of the intra-vertex v_i is greater than 1, shown in Eq.(1).

$$\bar{R}_{st} = R_{st} - \{\hat{P}_{st}\} - \{P_i | v_i \in P_i, \left|\Gamma^{in}(v_i)\right| > 1\} \tag{1}$$

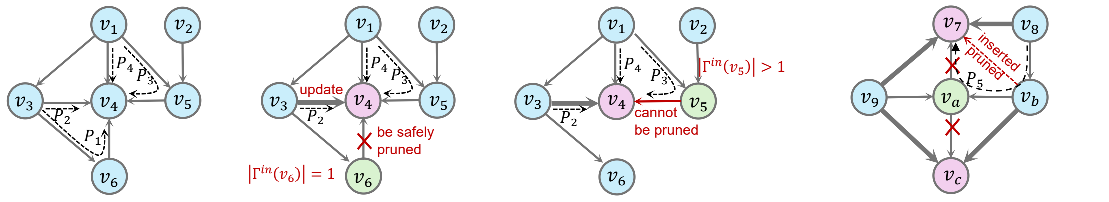
  That avoids possible loss of messages along any other path $P_z \notin R_{st}$. v_i and (v_s,v_t) actually form a triangle.
- Pruning the edge e_{it} that directly links to the target v_t for every path in $\bar{R}_{st}$.

We next use the concrete example in Fig.2(b) to explain our principle constraints. Given the pair of (v_3,v_4), we have redundant candidate set $R_{34} = \{P_1, P_2\}$. In particular, $\hat{P}_{34} = P_2$ is the compensation path. When some edges in other paths are pruned, the lost information originally delivered to v_4 can be merged into $\hat{P}_{34}$. That compensates for the loss without impact on other vertices, since the single edge $e_{34} \in \hat{P}_{34}$ directly links to v_4. $\bar{R}_{34}$ is then $\{P_1\}$. We then check which edge can be pruned for every path in $\bar{R}_{34}$. For the single path P_1, the intra-vertex v_6 has $|\Gamma^{in}(v_6)| = 1$. As shown in Fig. 2(b), we can safely remove the edge e_{64} in this path. The message loss can be delivered by the compensation path $\hat{P}_{14}$, by updating its weight (shown in Sec. IV-B). Afterwards, the number of total edges decreases by 1. Our focus is then moved to another pair (v_1,v_4). Similarly, we have $R_{14} = \{P_3, P_4\}$ and $\hat{P}_{14} = P_4$. However, by Eq.(1), $\bar{R}_{14} = \emptyset$ because the intra-vertex v_5 in the single remaining path P_3 has two in-neighbors, i.e., $|\Gamma^{in}(v_5)| = 2 > 1$. As shown in Fig. 2(c), in this case, if we prune e_{54}, we can compensate for the lost information delivered by $v_1 \rightarrow v_5 \rightarrow v_4$ through the preserved $\hat{P}_{14}$, but fail to cope with the loss from v_2, leading to incorrect computation.

B. *Correctness Compensation*

Pruning redundant paths can effectively reduce memory consumption for both edges and weights. However, this process destroys the graph topology, leading to information loss during vertex updates. To ensure correctness, we introduce compensation mechanisms. The core idea is to merge the weights of preserved paths/edges with those of pruned edges.

The correctness guarantee relies on an efficient traversal implementation. Typically, a traversal algorithm involves two fundamental operations, i.e., updating the value $\mathcal{V}(v_t)$ for each active vertex v_t, and generating a new message for an out-neighbor v_y based on the updated $\mathcal{V}(v_t)$ and the corresponding edge weight w_{ty}. We denote these operators as $\oplus$ and $\odot$, respectively. As established in Eq. (2), the prior work [42] has shown that at the j-th iteration, $\mathcal{V}^j(v_t)$ can be updated from its previous state $\mathcal{V}^{j-1}(v_t)$ by accumulating the change $\Delta\mathcal{V}^j(v_t)$, where $\Delta\mathcal{V}^j(v_t)$ is the accumulation of received messages



(a)The original graph (b)Safe pruning (c)Unsafe pruning (d) 3-hop implementation with α=1

Fig. 2: The compression framework where redundant paths are safely pruned

$\Delta\mathcal{V}^{j-1}(v_x) \odot w_{xt}$ from in-neighbors v_x. Now the message is generated based on the received change, instead of its state. This delta-based accumulative implementation improves efficiency, as inactive vertices can halt without repeatedly broadcasting messages. Such a scheme requires that '$\odot$' be *distributive* over '$\oplus$', and that '$\oplus$' satisfy the *commutative* and *associative* properties. Many algorithms like PPR and SSSP adhere to these mathematical requirements.

$$\begin{cases} \mathcal{V}^{j}(v_t) = \mathcal{V}^{j-1}(v_t) \oplus \Delta\mathcal{V}^{j}(v_t), \\ \Delta\mathcal{V}^{j+1}(v_t) = \sum \oplus\big(\Delta\mathcal{V}^{j}(v_x) \odot w_{xt}\big), \forall v_x \in \Gamma^{in}(v_t) \end{cases} \tag{2}$$

Following Eq. (2), Theorem 1 introduces our weight-merging compensation solution and ensures the correctness.

Theorem 1: Consider a triangle formed by vertices v_s, v_i, and v_t, where v_s links to both v_i and v_t, and v_i further links to v_t. Suppose $\Gamma^{\text{in}}(v_i) = \{v_s\}$ and '$\odot$' has the *commutative* property. In this case, the edge e_{it} and its weight w_{it} can be safely pruned, provided that the weight w_{st} is updated to a new value $\hat{w}_{st}$ according to Eq. (3).

$$\hat{w}_{st} = w_{st} \oplus \big(w_{si} \odot w_{it}\big) \tag{3}$$

Proof 1: Notably pruning the edge e_{it} disrupts the direct propagation of messages to vertex v_t. The proof therefore focuses on ensuring the correct accumulation of the required update messages $\Delta\mathcal{V}(v_t)$. According to Eq. (2), we have

$$\Delta\mathcal{V}(v_t) = \big(\Delta\mathcal{V}(v_s) \odot w_{st}\big) \oplus \big(\Delta\mathcal{V}(v_i) \odot w_{it}\big) \oplus \sum \oplus\big(\Delta\mathcal{V}(v_x) \odot w_{xt}\big), \forall v_x \in \big(\Gamma^{in}(v_t) - \{v_s, v_i\}\big)$$

. Since v_s is the only in-neighbor of v_i, i.e., $\Gamma^{\text{in}}(v_i) = \{v_s\}$, the term $\Delta\mathcal{V}(v_i)$ can be substituted with $\Delta\mathcal{V}(v_s) \odot w_{si}$. Using the *distributive* and *commutative* properties of the operators, we can rewrite $\Delta\mathcal{V}(v_t)$ as follows,

$$\begin{aligned} \Delta\mathcal{V}(v_t) &= \big(\Delta\mathcal{V}(v_s) \odot w_{st}\big) \oplus \Big(\big(\Delta\mathcal{V}(v_s) \odot w_{si}\big) \odot w_{it}\Big) \oplus ... \\ &= \Big(\Delta\mathcal{V}(v_s) \odot \big(w_{st} \oplus (w_{si} \odot w_{it})\big)\Big) \oplus ... \\ &= \big(\Delta\mathcal{V}(v_s) \odot \hat{w}_{st}\big) \oplus ... \end{aligned}$$

. This derivation confirms that merging the weights as described preserves the correctness of the message aggregation despite the removal of e_{it} and w_{it}.

Below we use three classic traversal algorithms to show how to merge messages by modifying weights, so as to correctly update the value of v_t, $\mathcal{V}(v_t)$.

<u>*SSSP*</u>. It computes the shortest distance from a given source to every other vertex. '$\oplus$' and '$\odot$' are the operators of 'min' and '+', respectively. $\Delta\mathcal{V}(v_t)$ is computed as below.

$$\begin{aligned} \Delta\mathcal{V}(v_t) &= \min\Big(\Delta\mathcal{V}(v_s) + \underline{w_{st}}, \Delta\mathcal{V}(v_s) + \underline{w_{si} + w_{it}}, ...\Big) \\ &= \min\Big(\Delta\mathcal{V}(v_s) + \underline{\min\big(w_{st}, w_{si} + w_{it}\big)}, ...\Big) \end{aligned}$$

<u>*PPR*</u>. It estimates the similarity between a pre-set source and every other vertex. Given a decay factor α, a vertex first sums up all received messages from in-neighbors and then transfers the (1-α) fraction of the sum as its own value, and the remain is broadcasted to out-neighbors based on corresponding weights. '$\oplus$' and '$\odot$' are the operators of '+' and '$\cdot$', respectively. $\Delta\mathcal{V}(v_t)$ is computed as below.

$$\begin{aligned} \Delta\mathcal{V}(v_t) &= \Big(\alpha\Delta\mathcal{V}(v_s) \cdot \underline{w_{st}} + \alpha\big(\alpha\Delta\mathcal{V}(v_s) \cdot \underline{w_{si}}\big) \cdot \underline{w_{it}} + ...\Big) \\ &= \alpha\Delta\mathcal{V}(v_s) \cdot \underline{\big(w_{st} + \alpha w_{si} w_{it}\big)} + ... \end{aligned}$$

<u>*Adsorption*</u>. It is a recommendation algorithm based on propagating labels. Labels as vertex values indicate the common features shared by entities. Resembling *PPR*, it iteratively propagates labels by Random Walking, until the distribution of labels is steady. Because the merging operation is the similar with *PPR*, we omit its correctness analysis for brevity.

In traversal algorithms, any vertex can serve as the source. However, pruning edge e_{it} would prevent the proper generation of the initial message if v_i becomes the source. To address this, a copy of e_{it} is retained and offloaded to disk, loaded only when required. These pruned edges are grouped by their source vertices and indexed by the corresponding source vertex IDs for fast data access. This design preserves memory savings while introducing negligible I/O overhead, as the edge is fetched at most once during the entire traversal. Taking v_6 in Fig.2(b) as an example, if chosen as a traversal source, the job immediately stalls since v_6 has no outgoing edges in the compressed graph. The pruned edge e_{64} is therefore loaded from disk to initialize message propagation. In SSSP, e_{64} is needed only for this initial activation. Later messages along P_3 are handled via the merged-weight path P_2. For algorithms where v_6 must broadcast repeatedly, e_{64} is kept in memory until the job finishes.

C. *Prioritized k-hop Implementation*

We now present the detailed implementation of our pruning algorithm. For a given source–target pair (v_s,v_t), Secs. IV-A and IV-B show that only the path $P_i : v_s \rightarrow v_i \rightarrow v_t$ in a triangle needs to be examined in order to prune edge e_{it}. For the intra-vertex $v_i \in P_i$, this implies that only its 2-hop neighbors (direct in- and out-neighbors) are considered. In practice, the pruning can be extended to k-hop neighbors to enable deeper compression.

1) Basic k-hop Pruning: We begin with the basic 2-hop implementation. We design an efficient vertex-centric procedure that avoids the expensive refinement step required in Eq.(1). For a given vertex v_i, we first enumerate each of its children v_y and then evaluate the subset condition given in Eq.(4).

$$\Gamma^{in}(v_i) \subseteq \Gamma^{in}(v_y), v_y \in \Gamma^{out}(v_i) \tag{4}$$

If the condition holds, then for every $v_x \in \Gamma^{in}(v_i)$, the edges e_{xy}, e_{xi} and e_{iy} form a triangle. All such triangles share the common edge e_{iy}. Once e_{iy} is pruned, we update w_{xy} by successively merging the weight w_{xi} of each edge e_{xi} together with e_{iy}, using Eq.(3). Thereafter, these messages can be correctly delivered along e_{xy}. We perform this examination for every vertex, thereby completing the pruning process of redundant paths for the entire input graph.

However, the condition in Eq.(4) is too strict that only a few vertices can satisfy it, which severely limits the number of edges that can be pruned. We therefore introduce a relaxed version of the condition, shown in Eq.(5).

$$\left|\Gamma^{in}(v_i) - \Gamma^{in}(v_y)\right| \leq \alpha, \alpha \geq 0, v_y \in \Gamma^{out}(v_i) \tag{5}$$

This means that at most α belong exclusively to $\Gamma^{in}(v_i)$ and not to $\Gamma^{in}(v_y)$. Under this relaxed constraint, e_{iy} can still be pruned even when Eq. (4) is not satisfied, but at expense of inserting a new edge e_{xy} for each such non-shared vertex v_x. Doing so ensures that the message along $v_x \rightarrow v_i \rightarrow v_y$ can be correctly delivered via e_{xy}. These newly inserted edges e_{xy} may themselves be pruned later when v_x is examined, ultimately yielding memory reduction. To avoid redundant insert-and-prune operations, we extend the examination from 2-hop to k-hop neighborhoods, checking whether ancestors of v_i's ancestors satisfy the pruning condition. Notably the maximum practical value of α is 1. In this case, even if e_{xy} cannot be pruned subsequently, the removal of e_{iy} can still offset its space cost, leaving the total memory cost unchanged.

Fig. 2(d) illustrates a 3-hop pruning example with $\alpha = 1$. When examining vertex v_a and its child v_c, we find that $\Gamma^{in}(v_a) = \{v_9, v_b\} \subset \Gamma^{in}(v_c) = \{v_9, v_a, v_b\}$. By Eq. (4), edge e_{ac} is pruned, and its weight is merged into w_{9c} and w_{bc}, respectively. For another child v_7, however, $v_b \in (\Gamma^{in}(v_a) - \Gamma^{in}(v_7))$. The strict condition in Eq. (4) does not permit pruning e_{a7}. Under the relaxed condition with $\alpha = 1$ in Eq. (5), e_{a7} can be pruned at cost of inserting a new edge e_{b7}. Although e_{a7} could later be pruned when v_b and its child v_7 are examined, this extra insert-prune cycle incurs runtime overhead. This overhead can be avoided by performing 3-hop examination along path P_5 from v_8 to v_7 (spanning 3 edges). In this case, e_{a7} is pruned directly and its weight is merged into the existing weight w_{87} of edge e_{87}, eliminating the need to insert and subsequently prune e_{b7}.

2) A Prioritized Variant: Another problem is that the frequent set-comparison operation in Eqs. (4) and (5) is very time consuming, but the target edge might not be eligible for pruning. We then give our prioritized examination implementation for runtime efficiency. In fact, the probability of successfully pruning an edge heavily depends on the in-degree of related vertices. Let φ_{xy} indicate the probability that e_{xy} exists between v_x and v_y. The probability of pruning an edge e_{iy} is $\Phi_{iy} = \prod_{v_x \in \Gamma^{in}(v_i)} \varphi_{xy}$ by the condition Eq.(4), or $\prod_{v_x \in (\Gamma^{in}(v_i) - \check{\Gamma}^{in}(v_i))} \varphi_{xy}$ by the relaxed condition Eq.(5). Here $\check{\Gamma}^{in}(v_i) = \Gamma^{in}(v_i) - \Gamma^{in}(v_y)$, indicating the subset of vertices that do not have edges linking to v_y. Clearly Φ_{iy} is low if v_i has a large number of in-neighbors. It is cost-ineffective to perform the set comparison for v_i. Our improved variant is to prioritize vertices in ascending order of their in-degrees, and then sequentially perform vertex-centric comparison and pruning operations, until the pruning benefit is low.

However, prioritizing all vertices in a fine-grained manner is runtime expensive. We thereby give a coarsen-grained alternative. We group vertices into several buckets with different in-degree ranges, and then process these buckets in ascending order of their average in-degrees. Another issue is the criterion of terminating the pruning process in advance. Our solution is a bucket-centric checking mechanism. After processing the x-th bucket B_x, we count the number of successfully pruned edges, $|\overline{B}_x|$. We then define the pruning ratio of B_x to the number of total edges, $|E|$. As expected, $\frac{|\overline{B}_x|}{|E|}$ gradually decreases with x. We terminate the pruning process when it is not greater than a pre-set threshold β. To determine β, Fig. 3 plots the ratio against the average in-degree, on many real-world datasets (see details in Table I). We can observe the consistent variation trend on all datasets. By default, we terminate computations when the average in-degree of a bucket is smaller than 15, i.e., $\beta \approx 3 \times 10^{-3}$. That strikes a good balance between compression and runtime efficiency.

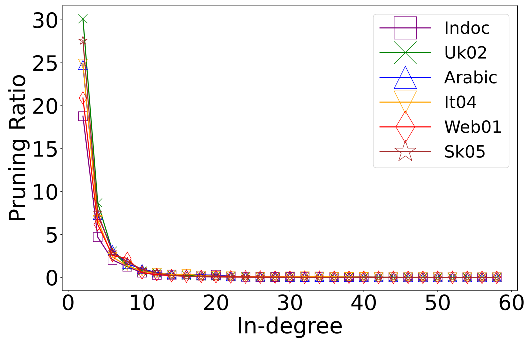


Fig. 3: Impact of the in-degree on the pruning ratio

Algorithm 1 outlines our prioritized pruning algorithm. We group vertices and then enumerate each bucket/group for pruning (Lines 1-3). In particular, given a vertex v_i, if redundant paths are detected based on Eq.(5) (Line 6), we update the compressed graph $\overline{G}$ by pruning edges as well as weights,

Algorithm 1: Prioritized Pruning Strategy

```
Input  : input graph $G$ and hyper-parameters $(\alpha,\beta)$
Output: compressed graph $\overline{G}$ and pruned data $(\overline{E}, \overline{W})$
1  $\mathcal{B} \leftarrow$ GroupbyInDegree($G$);
2  foreach $B_k \in \mathcal{B}$ do
3      $|\overline{B}_k| \leftarrow 0$, $\overline{G} \leftarrow G$;
4      foreach $v_i \in B_k$ do
5          foreach $v_y \in \Gamma^{out}(v_i)$ do
6              if hasRedundant($\Gamma^{in}(v_i)$,$\Gamma^{in}(v_y)$,$\alpha$)$\neq \emptyset$ then
7                  Update($\overline{G}$, $e_{iy}$, $w_{iy}$);
8                  $\overline{E} \leftarrow \overline{E} \cup \{e_{iy}\}$, $\overline{W} \leftarrow \overline{W} \cup \{w_{iy}\}$;
9                  $|\overline{B}_k|$++;
10     if $\frac{|\overline{B}_k|}{|E|} \leq \beta$ then
11         return $\overline{G}$ and $(\overline{E}, \overline{W})$;
```

and merging weights using Eq.(3) (Line 7). The pruned data are also recorded for correctness compensation (Line 8). We count the number of pruned edges and accordingly terminate compression in advance (Lines 9-11).

Last but not least, during compression, we logically mark edges and weights ought to be pruned, instead of performing physical behaviors. That avoids frequently modifying the CSR structure of graph data. The runtime delay of allocating and releasing memory thereby decreases. These marked data are physically pruned in a batch manner.

3) Complexity Analysis: Finally, we analyze the time and space complexities of our pruning procedure. For the basic k-hop pruning described in Eq. (5), each vertex v_i is paired with each of its out-neighbors, and then all in-neighbors of v_i are examined. This results in a time complexity of $O(d|E|)$ for 2-hop pruning, where d denotes the average in-degree. When extending to general k-hop pruning with $\alpha = 1$, increasing k by one expands the exploration to deeper in-neighbors for at most one currently examined in-neighbor, yielding a complexity of $O(kd|E|)$. Correspondingly, the space complexity is $O(|V| + |E| + kd)$. In practice, the prioritized implementation reduces both time and space costs by selectively enumerating vertices rather than processing all of them.

V. CPU-GPU Collaborative Processing

We now move our focus to the power utilization. Existing optimizations like CTA and memory coalescing can be integrated, but they are tailored only for a single traversal job. We thereby present new collaborative solution for multiple traversals. Below we first give the new design and then study some key issues to enhance the performance.

A. Load-aware Lightweight Migration

As described in Sec. III, the process of a graph traversal job can be divided into two stages. That is, the workload first quickly increases to the peak (**divergence**) and then gradually decreases (**convergence**). The latter often requires more than half of the total iterations. Now only a few threads of a GPU perform physical computations since a majority of vertices have already converged; others might be also busy due to the SIMD constraint, but they cannot boost the convergence speed. Worse, the high parallelism becomes cost-inefficient because synchronizing so many threads might easily offset the acceleration benefit. Besides, multiple traversal jobs are sequentially executed where the GPU accelerator continuously and periodically falls into the cost-benefit dilemma.

To resolve this dilemma, we give a new CPU-GPU collaborative processing design where the GPU is responsible for iterations with high workloads (more active vertices), while the CPU handles iterations with low workloads (fewer active vertices). All traversal jobs in the waiting queue are fed in a pipeline manner. The GPU thereby always processes the high workloads of every job, yielding a significantly improved utilization. As shown in Fig. 4, when the load of Job_1 decreases, it is migrated from GPUs to CPUs to complete remaining low-workload iterations. Meanwhile, Job_2 is scheduled onto GPUs.

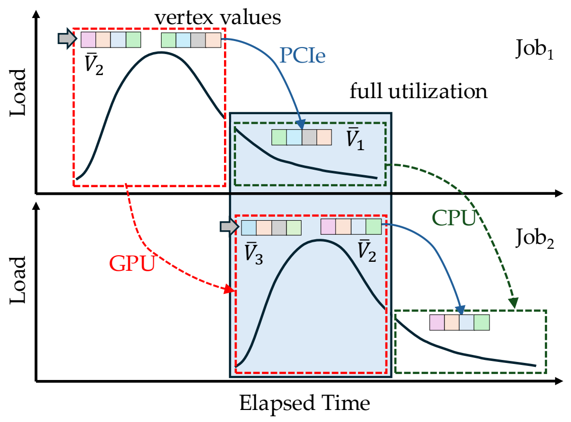


Fig. 4: The pipeline optimization for multiple queries

However, the limited PCIe bandwidth between GPUs and CPUs become the performance bottleneck for migration. Existing solutions focus on fine-grained scheduling among iterations within a single job, which frequently migrates a large volume of data between the two kinds of accelerators. Differently, our coarse-grained collaborative design primarily concerns on the scheduling among jobs. That provides opportunities to optimize both the migration frequency and the data volume, to avoid frequent interruptions on GPUs and reduce the runtime delay caused by migration. Below we give our lightweight optimizations.

One-pass migration. Given a traversal job, we observe a roughly monotonic increase and decrease about the workloads respectively in the divergence and the convergence stages. That means we should run it on CPUs at the very beginning and at the end of iterations where only a few vertices are involved; and on GPUs during the middle iterations. The number of migration is then reduced to only 2 in theory. Further, during the divergence stage, due to the prominent connection among vertices, the workload can quickly increase to the peak in a few iterations. The accumulated benefit achieved by running the job on CPUs is very marginal, and even cannot offset the additional migration cost. Thus, a job is launched on GPUs and migrated from GPUs to CPUs only once at the long convergence stage.

Computation migration. Different from swapping subgraphs or synchronizing immediate results between CPUs and GPUs, we migrate the computation logic and associate vertex values. To facilitate this lightweight design, we replicate the graph on two accelerators. Once the migration happens, we can quickly initialize the graph on CPUs and then seamlessly run remaining iterations. Clearly, CPUs are used to handle workloads that are cost-ineffective on GPUs. The memory scalability is solely addressed by compression in Sec. IV.

Our migration is lightweight but still not free, since the vertex values computed by GPUs need to be transferred. We thereby give an asynchronous implementation to overlap computations and data transferring operations. As shown in Fig. 4, right before migrating Job_1, we immediately make a copy of vertex values. The one of the two replicas is re-initialized and used for the next query, like $\overline{V}_2$ in Job_1 in Fig. 4; the other serves as a buffer for asynchronous migration. Note that CPUs are usually equipped with more memory resources than GPUs. We can buffer vertex values of many immigrated queries on CPUs. That provides enough flexibility to real-time tune the migration timing between GPUs and CPUs. We will give detailed descriptions in the next subsection.

B. Adaptive Migration Model

Under our pipeline collaborative framework, a straightforward implementation is to migrate the query currently running on GPUs onto CPUs once the latter is idle. That guarantees both of accelerators are always busy, to avoid power waste. However, for the first query job, its migration timing needs to be carefully determined, since now both accelerators are idle. For an over-early migration, Job_1 still has heavy workloads. The CPUs need to take a long time to process it, due to the limited compute power. As a negative impact, the successive query Job_2 on GPUs cannot be migrated in time, even though many vertices have converged and the parallel computations become cost-ineffective. Worse, when Job_1 finishes, Job_2 has few workloads left and hence it can be quickly completed even on CPUs. That in turn makes Job_3 migrated from GPUs to CPUs too early. The migration of all queries then falls into a vicious cycle where the utilization of GPUs is usually low.

We break such vicious cycle by determining a reasonable migration timing. The basic idea is that given a job Job_i, after migration at the j-th iteration, CPUs can complete the remaining workloads from the (j+1)-th iteration to the convergence point, with the same time that GPUs have already spent on processing the workloads during the total previous j iterations, i.e., $\mathcal{T}_{cpu}^{(j+1)\to\infty}(Job_i) = \mathcal{T}_{gpu}^{1\to j}(Job_i)$. We assume that the next job Job_{i+1} has the same behaviors with Job_i, i.e., $\mathcal{T}_{gpu}^{1\to j}(Job_i) = \mathcal{T}_{gpu}^{1\to j}(Job_{i+1})$, since they are run on the same underlying graph topology. Together, we have $\mathcal{T}_{cpu}^{(j+1)\to\infty}(Job_i) = \mathcal{T}_{gpu}^{1\to j}(Job_{i+1})$. That means the power of CPUs can be timely released by Job_i when Job_{i+1} should be migrated. In this way, we can avoid the underutilized usage of GPUs as much as possible.

Because the previous iterations are physically run on GPUs, $\mathcal{T}_{gpu}^{1\to j}(Job_i)$ is naturally available. Our focus is then to estimate $\mathcal{T}_{cpu}^{(j+1)\to\infty}(Job_i)$. Towards this end, we need to know the remaining workloads measured by the number of active vertices A^x at the x-th iteration, and the compute power difference of CPUs and GPUs. For the former, we found that although the number against the iteration is not a linear function in the entire iterations, the fitting error is very small in the convergence stage. We have validated this insight by extensively testing many traversal algorithms on real graphs, such as the result revealed in Fig. 1(a). Such a function $f(x)$ can be simulated by sampling two points in the most recent two iterations. For the latter, we need to know some real-time statics at the j-th iteration, including the runtime T_{gpu}^j, the runtime of synchronizing all threads S_{gpu}^j, and A^j. Then we can estimate the average time of processing a vertex on GPUs, i.e., T_{gpu}^j/A^j. This can be transferred as the estimation on CPUs, using a factor H_{gpu}/H_{cpu}. Here, H indicates the computing power of an accelerator. It is computed by the base frequency and the number of cores, both of which are available as the static hardware knowledge. The synchronization cost is also relative to this factor. Similarly, we can give an estimation on CPUs as $S_{gpu}^j \frac{H_{cpu}}{H_{gpu}}$. Eq.(6) finally gives the estimation of $\mathcal{T}_{cpu}^{(j+1)\to\infty}$. Accordingly, at every iteration, we compare the real $\mathcal{T}_{gpu}^{1\to j}$ against the estimated $\mathcal{T}_{cpu}^{(j+1)\to\infty}$. We launch the migration once $\mathcal{T}_{cpu}^{(j+1)\to\infty}(Job_i) \le \mathcal{T}_{gpu}^{1\to j}(Job_i)$ is satisfied.

$$\mathcal{T}_{cpu}^{(j+1)\to\infty} = \sum_{x=j+1}^{\infty} \Big(f(x) \cdot \frac{T_{gpu}^j}{A^j} \cdot \frac{H_{gpu}}{H_{cpu}} + S_{gpu}^i \frac{H_{cpu}}{H_{gpu}} \Big) \quad (6)$$

Algorithm 2 outlines our CPU-GPU collaborative migration process. Prior to executing any job, the compressed graph is replicated on both GPU and CPU (Line 1). Initially, a job is scheduled on the GPU. During each iteration, runtime statistics are collected to estimate whether the job should be migrated to the CPU (Lines 7-9). If so, the vertex-value vector of the current job is asynchronously transferred while the next job is being scheduled (Lines 10-12). The CPU device continuously receives migrated jobs and schedules them in FIFO order (Lines 14 and 17). The available CPU memory is updated in real time throughout this process (Lines 15 and 18).

C. Complexity Analysis

We proceed to analyze the time and space complexity of our migration optimization. Since collecting scalar statistics is efficient at runtime, the dominant cost stems from transferring the vertex value vector $\overline{V}$ of size $|V|$. As this vector is transferred only once per traversal, the time complexity is $O(|V|)$. The space complexity additionally includes the overhead of replicating the compressed graph structure, resulting in an overall complexity of $O(|V| + |E|)$.

VI. Experiment Evaluation

We implement our compression and collaboration techniques in a prototype system C^2*graph*. Resembling Pregel-like systems [4], [5], [42], C^2*graph* exposes the uniform

Algorithm 2: CPU-GPU Collaboration Strategy

```
Input : compressed graph $\overline{G}$, hardware statistics $H_{\text{gpu}}$, $H_{\text{cpu}}$ and
        available CPU memory $\mathcal{M}_{\text{cpu}}$, and a queue of jobs $Q_{\text{wait}}$
Replicating $\overline{G}$ on GPU and CPU devices;
foreach $Job_i \in Q_{\text{wait}}$ do
    Scheduling $Job_i$ onto GPU;
    GPU Device:
    foreach iteration $j$ of $Job_i$ do
        Updating vertices and propagating messages by Eq. (2);
        Estimating $\mathcal{T}_{cpu}^{(j+1)\to\infty}(Job_i)$ by Eq. (6);
        if $\mathcal{T}_{cpu}^{(j+1)\to\infty}(Job_i) \leq \mathcal{T}_{gpu}^{1\to j}(Job_i)$ & $\mathcal{M}_{\text{cpu}} \geq$
         EstMem($\overline{V}_i$) then
            Allocating GPU memory for $\overline{V}_{i+1}$;
            Migrating $\overline{V}_i$ from GPU to CPU asynchronously;
            Break;
CPU Device:
Adding a received $Job_i$ into an FIFO queue $Q_{\text{cpu}}$ at any time;
$\mathcal{M}_{\text{cpu}} \leftarrow \mathcal{M}_{\text{cpu}} - \text{EstMem}(\overline{V}_i)$;
foreach $Job_x \in Q_{\text{cpu}}$ do
    Computing $Job_x$ until convergence;
    $\mathcal{M}_{\text{cpu}} \leftarrow \mathcal{M}_{\text{cpu}} + \text{EstMem}(\overline{V}_x)$;
```

APIs to insulate programmers from tedious low-level details. End-users can easily program graph traversal algorithms by implementing the two operators '$\oplus$' and '$\odot$', and initializing the delta-value $\Delta\mathcal{V}(v_t)$ for each vertex v_t.

In general, *C^2graph* performs graph traversals on a compressed/pruned graph in a memory-efficient manner. Pruned edges and their corresponding weights are offloaded to disk and thus consume no main memory. As explained in Sec. IV-B, each disk-resident edge is loaded at most once during a given traversal, making the resulting I/O overhead negligible. Furthermore, *C^2graph* enables efficient CPU-GPU collaboration by maintaining a replica of the compressed graph in CPU memory. This replication does not occupy GPU memory capacity, thereby providing a foundation for job migration and hybrid scheduling. Below we evaluate these abilities mentioned above.

A. Experimental Setup

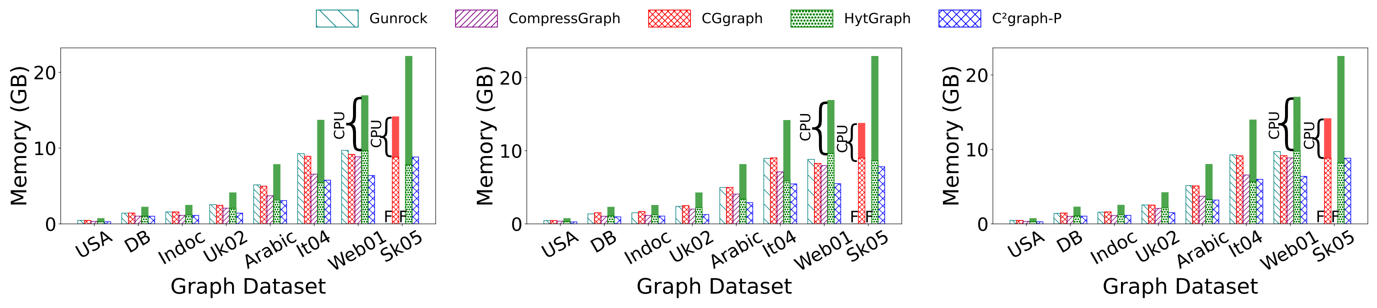
1) Hardware, Benchmarks and Datasets: All experiments are conducted on a server equipped with a NVIDIA GeForce RTX3080 GPU (8704 CUDA Cores, 1.71GHz, 10GB GDDR6X memory) and an Intel(R) Core(TM) i9-10900K CPU (10 Cores, 3.70GHz, 64GB of DDR4 memory), connected by PCIe 3.0. We test three graph traversal algorithms as benchmarks, including SSSP, PPR and Adsorption, as mentioned in Sec. IV-B, to validate the adaptability of our proposals. Table I summarizes the statistics of real-world graph datasets from open-source libraries[1][2][3][4][5]. The average degree varies from 0.7 to 39, indicating different redundancies. Notably in many real-world graphs, edge weights are often unavailable due to privacy considerations. Following existing studies [23], [43], [44], we first partition vertices into 20 communities and then assign a weight to intra-community edges that is twice that of inter-community edges. Besides, following the setting of [45], all duplicated edges within an adjacency list and self-loops are removed. In addition, we run a two-layer Graph Convolutional Network (GCN) with 256 hidden dimensions on the feature graph OgbArxiv to evaluate the effectiveness of our method on complex graph analysis tasks. Graphs are stored in Compression Sparse Row (CSR) to facilitate data access.

[1]http://snap.stanford.edu/data/index.html
[2]http://law.di.unimi.it/datasets.php
[3]http://konect.cc/networks/
[4]https://ogb.stanford.edu/docs/graphprop/
[5]https://toreopsahl.com/datasets/

TABLE I: Real-world graph datasets (M: million, B: billion)

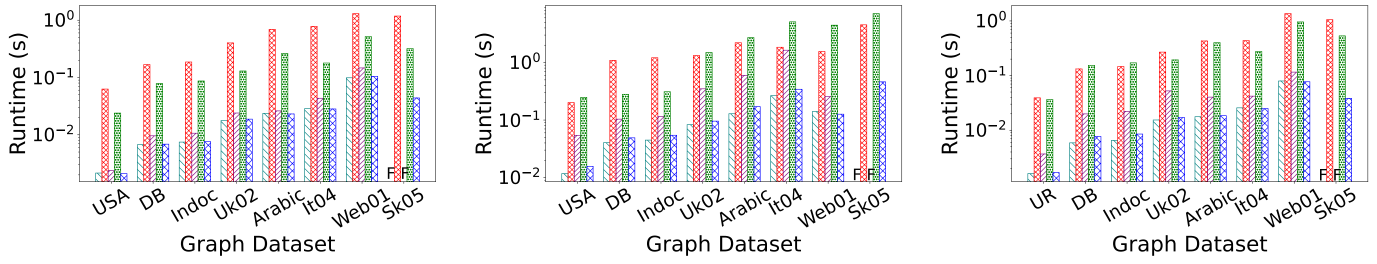
Graphs	$\lvert V\rvert$	$\lvert E\rvert$	**Avg. Deg.**	**Weighted**	**Size(CSR)**
Enron	0.09M	0.32M	3.7	Yes	2.8MB
Slash	0.08M	0.52M	6.5	Yes	4.2MB
Higgs	0.47M	0.33M	0.7	Yes	4.2MB
DBpedia	18M	136M	7.5	Yes	1.1GB
USARoad	24M	58M	2.4	Yes	532MB
OgbArx	0.17M	1.2M	6.9	No	9.5MB
Amazon	0.74M	5.2M	7.0	No	42MB
Skitter	1.7M	11M	6.5	No	91MB
Cite	6.0M	16M	2.7	No	149MB
Indoc	7.4M	194M	26	No	1.4GB
Uk02	19M	298M	16	No	2.2GB
Arabic	23M	640M	28	No	4.7GB
It04	41M	1.15B	27	No	8.5GB
Web01	118M	1.02B	8.6	No	7.9GB
Sk05	51M	1.95B	39	No	14.4GB

2) Compared Solutions: We create two branches for our *C^2graph* framework. The basic one has only the Pruning compression component, termed as **C^2graph-P** (Sec. IV). It works for a single traversal query. The improved variant additionally integrates the lightweight Migration optimization tailored for multiple traversals, termed as **C^2graph-PM** (Sec. V). **CompressGraph** [14] presents the rule-based compression technique by replacing common edges. Till now, it achieves the best edge compression ratio. Notably its up-to-date variant Laconic [32] is excluded, because its focus is making a compromise between compression runtime and compression ratio, that is beyond the scope of our primarily concerned memory scalability. Another competitor is the most recent CPU-GPU collaborative system **CGgraph** where the CPU memory is used to provide more storage resources. It particularly makes "sink vertices" with only one in-neighbor separate from the original graph, and updates them once after other vertices normally converge. That reduces the memory requirement during normal iterations. Besides, it also utilizes the CPU power by migrating the computation focus from GPUs to CPUs if necessary, for better runtime efficiency. **HytGraph** [13] is another recently released framework that can smartly choose communication-efficient data transfer manner between CPU and GPU. Finally, we include results from a well-known GPU-based graph processing library **Gunrock** that has integrated classical optimizations. It works as a baseline to measure the scalability of compared solutions in terms of runtime latency and memory reduction. Notably, we use the open-source implementation of *Gunrock* and *CGgraph*.



(a) PPR (b) SSSP (c) Adsorption

Fig. 5: Comparison on the memory consumption (a single query)



(a) PPR (b) SSSP (c) Adsorption

Fig. 6: Comparison on the computation runtime (a single query)

CompressGraph consists of compression and computation modules, but the latter cannot cope with weighted traversals. We thereby rewrite it by ourselves.

3) Testbeds: We run all benchmarking algorithms until convergence, i.e., all vertices become inactive. For PPR and Adsorption, the decay factor is set as $\alpha = 0.2$ as usual. Given a query, we primarily concern on two metrics during iterations, including the memory consumption and the iterative computation runtime (without compression runtime). The reported metrics are the averages over 10 runs for each combination of benchmarks and graphs, with randomly selected sources for each traversal. Since small graphs are not highly sensitive to GPU utilization, we report only their space performance. For large graphs (with size greater than 500 MB), we report both space and time metrics. Notably for some GPU-only competitors, we use the symbol "F" to indicate an Out-Of-Memory (OOM) Failure run when processing large graphs.

B. Overall Comparison of A Single Traversal Query

1) Results on Large Graphs: We exclude C^2*graph-PM* since it is tailored for multiple traversals. Fig. 5 and Fig. 6 reveal that C^2*graph-P* has the best time-space performance.

Fig. 5 reports the memory consumption of different competitors. *Gunrock* does not employ any memory-related optimization and hence has the worst performance. *CGgraph* outperforms the naive *Gunrock* due to the separation of sink vertices. However, the fraction of sink vertices is less significant. The memory reduction is thereby marginal (at most 8.2% for *SSSP* on *Sk05*). Further, *CompressGraph* largely reduces the memory consumption by deeply replacing all possible common edges with a single copy. Compared with *Gunrock*, the reduction is 22.1% on average. However, identifying target ids of common edges resorts to strict string matching operations, that cannot work for edge weights with distinct values. The compressed graph is still large. We even observe the OOM error on *Sk05*. Differently, our C^2*graph-P* can prune both edges and associated weights. The reduction factor against *Gunrock* is increased to 47.9% at most (*SSSP* on *Sk05*, and 36.5% on average). Even compared with *CompressGraph*, the factor is still up to 36.6% (*SSSP* over *Uk02*, 20.1% on average). As a result, it can handle all graphs only using GPUs. Notably *CGgraph* and *HytGraph* can also run computations on *Sk05*. This is due to the usage of additional CPU memory resources (as marked in Fig. 5). For *CGgraph*, the total memory consumption is up to 1.9× larger than that used by our C^2*graph-P*. For *HytGraph*, the factor is 2.6× at most.

Fig. 6 explores the impact of compression and/or CPU-GPU collaboration on runtime. *CGgraph* generally has the most significant runtime penalty because of the compute bottleneck of CPU under a skewed subgraph partitioning result between GPU and CPU[6]. The frequent interruption on GPUs caused by synchronizing messages also contributes to runtime delay. *HytGraph* mitigates these issues by its dynamic scheduling among multiple optimization policies, yielding up to 4.3× speedup (*PPR* on *It04*). However, this scheduling is very sensitive to its built-in runtime estimation result. The exception

[6]We run *CGgraph* by its default setting.

is *SSSP*. The unreasonable scheduling decision even makes it 2.8× slower than *CGgraph* (*SSSP* on *Web01*). *CompressGraph* avoids this complex CPU-GPU management but it increases the diameter of graph topology because of the newly added adjacency lists. Compared with *Gunrock*, the delay is still up to 83.7% (*SSSP* over *It04*, 47.6% on average). By contrast, our proposal is comparable to and even better than *Gunrock* due to the physically pruned edges and weights. It runs 2.3× faster than *CompressGraph* on average. The speedup factor compared with *CompressGraph* is up to 4.7×(*SSSP* on *It04*). For *CGgraph* and *HytGraph*, the factors are up to 29.8× and 25.1× respectively.

2) Results on Small Graphs: We next present the space savings achieved by our C^2*graph-P* and *CompressGraph* on small, low-redundancy graphs. The results are modest and comparable. On average, C^2*graph-P* achieves a 6.6% reduction (Enron: 4.7%, Slash: 3.4%, Higgs: 2.7%, Amazon: 6.0%, Skitter 9.8%, Cite: 12.8%), while *CompressGraph* achieves 6.0% (Enron: 8.8%, Slash: 2.7%, Higgs: 1.2%, Amazon: 5.3%, Skitter 13.7%, Cite: 4.4%). The performance gap narrows due to limited compression opportunities. Notably, both methods show almost no compression on the sparse *USARoad* graph. Besides, on the feature graph *OgbArx*, C^2*graph-P* reduces memory costs by 19.4%, outperforming *CompressGraph* (5.8%). The GCN benchmark runs in approximately 90 seconds on our system.

C. Overall Comparison of Multiple Traversal Queries

We next test the performance when processing multiple traversal queries. As validated in previous experiments, our C^2*graph-P* performs the best. We thereby exclude other competitors for brevity. In particular, we enumerate all possible combinations of benchmarking algorithms and graph datasets. For each case, we vary the number of query tasks from 100 to 500, and then report the distinctly total computation runtime. Our C^2*graph-P* and C^2*graph-PM* are involved in tests.

Fig. 7 plots runtime versus the number of queries. The results of *SSSP* and *Adsorption* are omit for manuscript space, since they are similar with those of *PPR*. We observe the prominent scalability in every case. Both of our proposals have linear runtime increase with the query number. However, the slope of the trend w.r.t. C^2*graph-PM* is less significant, due to the smart migration between CPUs and GPUs. C^2*graph-PM* thereby runs up to 1.9X times faster than C^2*graph-P* solely running on GPUs (1.7X on average). In particular, the increasing number of queries provides more opportunities for our adaptive model to fine-tuning the practical migration timing. CPUs and GPUs can work together in a seamless manner. Thus, the runtime gap between our two proposals nearly linearly increases.

Fig. 8 explores the impact of different migration timing, to validate the effectiveness of our adaptive migration model. We manually vary the specific iteration where the migration happens, and then depict the computation runtime. For all of the three benchmarking algorithms, we find that the runtime gradually decreases and then increases as expected. At the extreme case where the query is migrated at the first iteration, C^2*graph-PM* degrades to a CPU-based C^2*graph-P*. The low compute power definitely heavily limits the overall throughput. Another extreme case is that the migration happens at the convergence point. Now CPUs are always idle and GPUs suffer from underutilization at the convergence stage. The throughput is also far from ideal. Note that the optimal timing varies with benchmarks and datasets. However, our adaptive model can always roughly hit the "sweet spot" (marked by the box in Fig. 8). It helps end-users to gain nearly optimal performance while insulating them from tedious details.

Fig. 8(d) particularly illustrates the GPU power utilization. The workload is measured by the number of active vertices and generated messages from them. For C^2*graph-P*, a convergence phase of a query job emerges from 710ms to 1880ms. The reduced workloads leave most of GPU threads idle. C^2*graph-PM* promptly offloads a query to CPU once its workloads diminish, while simultaneously launching a new query and running it on GPU. This strategy significantly mitigates resource underutilization.

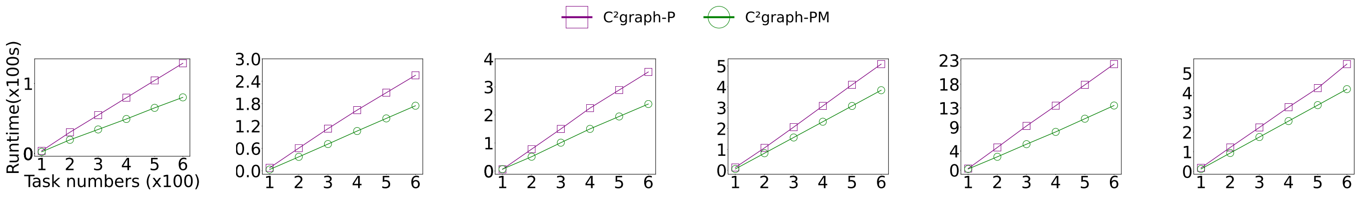
D. Comparison of the Compression Complexity

We now present the time-space complexity analysis of the compression solutions for end-to-end comparisons. Notably, when the requirement exceeds the capacity of GPU memory, *CGgraph* selects some vertices and then retains them in CPU memory. This selection operation is also involved in our comparison, as it alleviates the GPU memory bottleneck. Table II summarizes the runtime results. We evaluate two variants of C^2*graph*, including the basic implementation C^2*-basic* and the optimized prioritized version C^2*-pri* (Sec. IV-C). *CGgraph* performs selection on the large graph *Sk05*. It takes vertex degrees and BFS-based topology sorting results into consideration. Compared to *CompressGraph*, this is runtime efficient but needs to cache many sorting results, consuming a lot of memory resources as shown in Table III. Unlike *CompressGraph*, which processes only graph topology, C^2*-basic* additionally loads edge weights to support weight merging. This increases time-space overheads. *CompressGraph* also incurs costs due to its recursive compression rule, which requires repeated adjacency-list scans. In contrast, C^2*-pri* reduces runtime by up to 69.0% (*Sk05*) and 32.7% on average versus C^2*-basic*, due to early termination. When against *CompressGraph*, the reaches up to 62.0% (*Web01*).

TABLE II: Comparison on the compression runtime (secs)

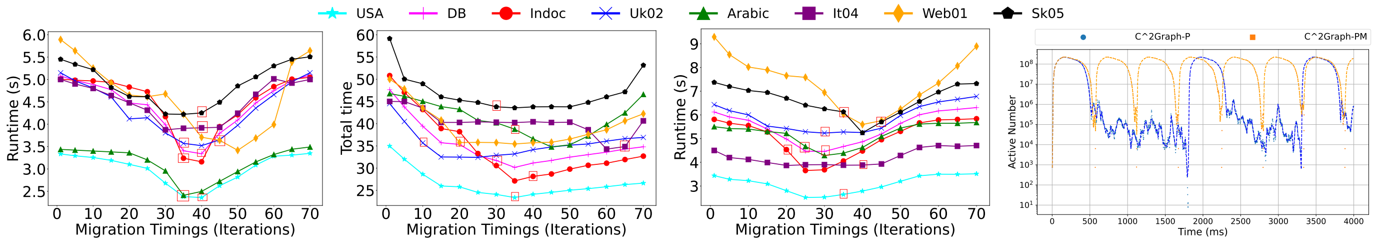
Datasets	**CGgraph**	**CompressGraph**	**C^2-basic**	**C^2-pri**
Indoc	–	85	54	40
Uk02	–	166	86	71
Arabic	–	302	198	189
It04	–	652	1089	380
Web01	–	935	413	355
Sk05	304	1232	2745	852

Table III presents the analysis of space complexity. Our proposals exhibit high memory requirements due to their need to process edge weights. In contrast, *CompressGraph* operates



(a) Indoc (b) Uk02 (c) Arabic (d) It04 (e) Web01 (f) Sk05

Fig. 7: Comparison on the total computation runtime (multiple queries, PPR)

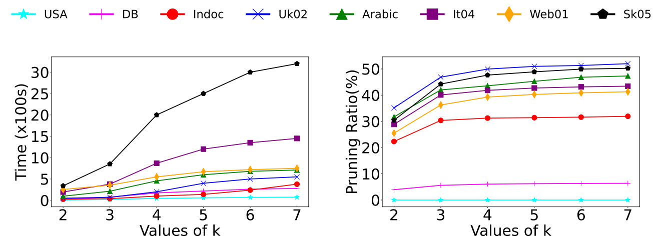


(a) PPR (b) SSSP (c) Adsorption (d) Workload variation

Fig. 8: Impact of different migration timings on the total computation runtime. Sub-figure (d) particularly demonstrates the workload variation on GPU (Arabic, SSSP).

TABLE III: Comparison on the compression memory (GB)

Datasets	CGgraph	CompressGraph	C^2-basic	C^2-pri
Indoc	–	4.00	5.50	5.01
Uk02	–	8.22	9.04	8.47
Arabic	–	13.23	19.28	17.02
It04	–	24.51	33.53	30.66
Web01	–	44.45	43.52	33.06
Sk05	46.70	37.12	55.08	50.02



(a) Impact on runtime (b) Impact on pruning ratio

Fig. 9: Impact of the k values in *C*^{2}*-pri*

solely on graph topology. However, its compression algorithm exhaustively enumerates edge sets and substitutes common patterns, which can become memory-intensive when common sets are infrequent (i.e., there are a large amount of disjoint edge sets). This overhead may surpass the memory cost of weight storage, as demonstrated by *Web01*—a graph with particularly low compression efficiency (see Fig. 5).

Finally we explore the impact of the hyperparameter k in our *C*^{2}*-pri* solution. As k increases from 2 to 7, the compression runtime rises due to the examination of more in-neighbors, while the number of pruned edges also grows accordingly. We observe that $k = 3$ represents a favorable trade-off between the additional runtime cost and the gain in pruning ratio. In all our experiments, we therefore set $k = 3$ and $\alpha = 1$ as the default configuration.

VII. Conclusion

This paper addresses time and space inefficiencies in traversal algorithms on weighted graphs, when running them on the modern GPUs with limited memory resources and specific SIMD hardware design. We propose a novel compression algorithm, *C*^{2}*graph-P*, which reduces memory usage by pruning redundant paths and compressing edge weights, while ensuring traversal correctness. To handle dynamic workloads, we introduce *C*^{2}*graph-PM*, a CPU-GPU collaborative framework that enhances computational efficiency through asynchronous query execution. Extensive experiments validate that our techniques can significantly reduce the memory consumption and increase the throughput.

Acknowledgements

This work was supported by the National Key R&D Program of China (2023YFB4503600), the National Natural Science Foundation of China (62572139, U25A20424, 62572108, U2436208, and 62372129), the Key R&D Program of Shandong (2025CXPT082), the National Key R&D Plan (2023YFB3107300), Guangdong S&T Program (2024B0101010002), the Project of Guangdong Key Laboratory of Industrial Control System Security (2024B1212020010), and the Key Laboratory of Computing Power Network and Information Security, Ministry of Education under Grant No. 2024ZD016.

References

[1] S. Bian, Q. Guo, S. Wang, and J. X. Yu, "Efficient algorithms for budgeted influence maximization on massive social networks," *Proc. VLDB Endow.*, vol. 13, no. 9, pp. 1498–1510, 2020.

[2] A. Al-Baghdadi and X. Lian, "Topic-based community search over spatial-social networks," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2104–2117, 2020.

[3] M. Then, M. Kaufmann, F. Chirigati, T. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo, "The more the merrier: Efficient multi-source graph traversal," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 449–460, 2014.

[4] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2010, pp. 135–146.

[5] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu, "Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing," in *Proc. of SIGMOD*. ACM, 2016, pp. 479–494.

[6] M. Kim, K. An, H. Park, H. Seo, and J. Kim, "GTS: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 447–461.

[7] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-gpu-memory graph processing," in *EuroSys '20: Fifteenth EuroSys Conference 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 12:1–12:16.

[8] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a GPU," in *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017*. IEEE Computer Society, 2017, pp. 233–245.

[9] P. Gera, H. Kim, P. Sao, H. Kim, and D. A. Bader, "Traversing large graphs on gpus with unified memory," *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 1119–1133, 2020.

[10] Y. Zhang, D. Peng, X. Liao, H. Jin, H. Liu, L. Gu, and B. He, "Largegraph: An efficient dependency-aware gpu-accelerated large-scale graph processing," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, pp. 58:1–58:24, 2021.

[11] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward unified-memory-efficient high-performance graph processing on GPU," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 2, pp. 22:1–22:25, 2021.

[12] S. Li, R. Tang, J. Zhu, Z. Zhao, X. Gong, W. Wang, J. Zhang, and P. Yew, "Liberator: A data reuse framework for out-of-memory graph computing on gpus," *IEEE Trans. Parallel Distributed Syst.*, vol. 34, no. 6, pp. 1954–1967, 2023.

[13] Q. Wang, X. Ai, Y. Yan, S. Gong, Y. Zhang, J. Chen, and G. Yu, "Towards communication-efficient out-of-core graph processing on the GPU," *IEEE Trans. Parallel Distributed Syst.*, vol. 36, no. 5, pp. 961–976, 2025.

[14] Z. Chen, F. Zhang, J. Guan, J. Zhai, X. Shen, H. Zhang, W. Shu, and X. Du, "Compressgraph: Efficient parallel graph analytics with rule-based compression," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 4:1–4:31, 2023.

[15] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *Proceedings of the International Conference on Web Search and Web Data Mining, WSDM 2008*, M. Najork, A. Z. Broder, and S. Chakrabarti, Eds. ACM, 2008, pp. 95–106.

[16] P. Boldi and S. Vigna, "The webgraph framework I: compression techniques," in *Proceedings of the 13th international conference on World Wide Web, WWW 2004*, S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, Eds. ACM, 2004, pp. 595–602.

[17] M. Sha, Y. Li, and K. Tan, "Gpu-based graph traversal on compressed graphs," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 775–792.

[18] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, V. Sarkar and L. Rauchwerger, Eds. ACM, 2017, pp. 235–248.

[19] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-gpu graph analytics," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS*. IEEE Computer Society, 2017, pp. 479–490.

[20] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 38–52.

[21] W. Lin, X. Xiao, X. Xie, and X. Li, "Network motif discovery: A GPU approach," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 3, pp. 513–528, 2017.

[22] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. of 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 17–30.

[23] P. Cui, H. Liu, B. Tang, and Y. Yuan, "Cggraph: An ultra-fast graph processing system on modern commodity CPU-GPU co-processor," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1405–1417, 2024.

[24] H. Liu, H. H. Huang, and Y. Hu, "ibfs: Concurrent breadth-first search on gpus," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 403–416.

[25] Z. Wang, Y. Gu, Y. Bao, G. Yu, J. X. Yu, and Z. Wei, "Hgraph: I/o-efficient distributed and iterative graph computing by hybrid pushing/pulling," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 5, pp. 1973–1987, 2021.

[26] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Trans. Parallel Distributed Syst.*, vol. 25, no. 6, pp. 1543–1552, 2014.

[27] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: processing large-scale graphs on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015*, J. Kern and J. S. Vetter, Eds. ACM, 2015, pp. 28:1–28:12.

[28] R. Tang, Z. Zhao, K. Wang, X. Gong, J. Zhang, W. Wang, and P. Yew, "Ascetic: Enhancing cross-iterations data efficiency in out-of-memory graph processing on gpus," in *ICPP 2021: 50th International Conference on Parallel Processing*, X. Sun, S. Shende, L. V. Kalé, and Y. Chen, Eds. ACM, 2021, pp. 41:1–41:10.

[29] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 49–63.

[30] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proc. IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.

[31] F. Claude and G. Navarro, "Fast and compact web graph representations," *ACM Trans. Web*, vol. 4, no. 4, pp. 16:1–16:31, 2010.

[32] Q. Xu, J. Yang, F. Zhang, Z. Chen, J. Guan, K. Chen, J. Fan, Y. Shen, K. Yang, Y. Zhang, and X. Du, "Improving graph compression for efficient resource-constrained graph analytics," *Proc. VLDB Endow.*, vol. 17, no. 9, pp. 2212–2226, 2024.

[33] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International Conference on World Wide Web, WWW 2011*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM, 2011, pp. 587–596.

[34] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita, "Compressing graphs and indexes with recursive graph bisection," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, Eds. ACM, 2016, pp. 1535–1544.

[35] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *2015 Data Compression Conference, DCC 2015*, A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer, Eds. IEEE, 2015, pp. 403–412.

[36] P. Gera and H. Kim, "Traversing large compressed graphs on gpus," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023*. IEEE, 2023, pp. 25–35.

[37] A. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: a total order approach," in *Proc. of SIGMOD*. ACM, 2014, pp. 1323–1334.
[38] X. Chen, Y. Peng, S. Wang, and J. X. Yuo, "Dlcr:efficient indexing for label-constrained reachability queries on large dynamic graphs," *Proc. VLDB Endow.*, vol. 15, no. 8, pp. 1645–1657, 2022.
[39] C. Zhang, A. Bonifati, H. Kapp, V. I. Haprian, and J.-P. Lozi, "A reachability index for recursive label-concatenated graph queries," in *Proc. of ICDE*. IEEE, 2023, pp. 67–81.
[40] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 577–588, 2014.
[41] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for youtube: taking random walks through the view graph," in *Proceedings of the 17th International Conference on World Wide Web, WWW 2008*. ACM, 2008, pp. 895–904.
[42] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Trans. Parallel Distributed Syst.*, vol. 25, no. 8, pp. 2091–2100, 2014.
[43] D. Duan, Y. Li, Y. Jin, and Z. Lu, "Community mining on dynamic weighted directed graphs," in *Proc. of CIKM*, J. Wang, S. Zhou, and D. Zhang, Eds. ACM, 2009, pp. 11–18.
[44] S. Günnemann and T. Seidl, "Subgraph mining on directed and weighted graphs," in *Proc. of PAKDD*, ser. Lecture Notes in Computer Science, vol. 6119. Springer, 2010, pp. 133–146.
[45] D. A. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner, "Benchmarking for graph clustering and partitioning," in *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*, R. Alhajj and J. G. Rokne, Eds. Springer, 2018.