Towards Communication-Efficient Out-of-Core Graph Processing on the GPU

Qiange Wang, Xin Ai, Yongze Yan, Shufeng Gong, Yanfeng Zhang, Jing Chen, Ge Yu, *Senior Member, IEEE*

***Abstract*—The key performance bottleneck of large-scale graph processing on memory-limited GPUs is the host-GPU graph data transfer. Existing GPU-accelerated graph processing frameworks address this issue by managing the active subgraph transfer at runtime. Some frameworks adopt explicit transfer management approaches based on explicit memory copy with filter or compaction. In contrast, others adopt implicit transfer management approaches based on on-demand accesses with the zero-copy mechanism or unified virtual memory. Having made intensive analysis, we find that as the active vertices evolve, the performance of the two approaches varies in different workloads. Due to heavy redundant data transfers, high CPU compaction overhead, or low bandwidth utilization, adopting a single approach often results in suboptimal performance. Moreover, these methods lack effective cache management methods to address the irregular and sparse memory access pattern of graph processing. In this work, we propose a hybrid transfer management approach that takes the merits of both two transfer approaches at runtime. Moreover, we present an efficient vertex-centric graph caching framework that minimizes CPU-GPU communication by caching frequently accessed graph data at runtime. Based on these techniques, we present HytGraph, a GPU-accelerated graph processing framework, which is empowered by a set of effective task-scheduling optimizations to improve performance. Experiments on real-world and synthetic graphs show that HytGraph achieves average speedups of 2.5×, 5.0×, and 2.0× compared to the state-of-the-art GPU-accelerated graph processing systems, Grus, Subway, and EMOGI, respectively.**

***Index Terms*—GPU, Graph processing, Communication reduction, Transfer management, Out-of-core processing**

I. Introduction

HIGH-performance graph processing is crucial for real-world graph applications, such as geo-information mining and social network analysis. Compared with CPU-based graph processing frameworks, GPU-based graph processing frameworks attract increased attention for their ability to leverage GPU's high memory bandwidth and massive parallelism [23], [37], [42], [46], [51]. However, the limited memory capacity of GPUs presents challenges in handling large-scale real-world graphs, especially when their sizes exceed the available GPU memory.

Recently, research [13], [14], [28], [31], [38], [39], [43], [50] has been directed toward developing GPU-accelerated graph processing systems that leverage high-performance GPU processing and the substantial memory capacity of the CPU. Similar to that of out-of-core graph processing [25], [36], [41], [52] on the CPUs, GPU-accelerated graph processing suffers from low GPU utilization caused by extensive CPU-to-GPU data movement overhead. Accessing data from the CPU needs data migration over the low-bandwidth PCIe interconnect (up to 32GB/s for PCIe 4.0), which can be an order of magnitude slower than the global memory access. Moreover, advances in PCIe interconnects have not effectively bridged the bandwidth gap, as the memory bandwidth of GPUs also increases simultaneously [34], [35]. This highlights the necessity of optimizing GPU-CPU data transfer.

Existing GPU-accelerated frameworks [13], [18], [31], [38], [39], [43], [50] mitigate data communication by tracking the evolving active vertices throughout iterative computation. In vertex-centric graph processing, computations are executed in a sequence of iterations, each processing vertices updated and marked as active in the prior iteration (i.e., active vertices), updating the out-going neighbors, and marking any neighbors with modified values as active vertices for the next iteration. In this process, it is necessary to access the edge data associated with active vertices (i.e., active subgraphs) [50]. Following existing systems [13], [18], [31], [38], [39], [43], [50], we assume that data related to vertices (such as value, neighbor index, and activity status) can reside in the GPU memory, the edge-associated data (including neighbor identities and weights) is entirely accommodated in the CPU memory, and, subgraphs comprising active edges are dynamically transferred to the GPU during iterative processing.

According to the way of reducing CPU-GPU active subgraph transfer, existing systems can be classified into two categories: **<u>Ex</u>plicit <u>T</u>ransfer <u>M</u>anagement (ExpTM)**-based frameworks [18], [38]–[40], [50] and **<u>Imp</u>licit <u>T</u>ransfer <u>M</u>anagement (ImpTM)**-based frameworks [13], [31], [43]. In ExpTM-based frameworks, active subgraph communication is managed by the programmers. The oversized graph is split into small graph partitions, each of which can fit within the GPU memory. Before these subgraphs are transferred to the GPU via the explicit CUDA memory copy (`cudaMemcpy`), they must be processed by a CPU-based redundancy removal module to eliminate inactive edges. Depending on the operation mode, these approaches can be either light-weight filter-based [18], [39] or heavy-weight compaction-based [38], [50].

Recently, ImpTM-based approaches that circumvent the explicit management of data movements for active subgraphs have emerged [13], [31], [43]. ImpTM-based frameworks utilize Unified Virtual Addressing (UVA) technique to map CPU and GPU to the same memory address, allowing GPUs to directly access the required active edges in the CPU [4], [5], [13], [31]. Compared with ExpTM, ImpTM requires less engineering efforts, allowing users to directly extend a single GPU framework into an out-of-core one by managing communication through unified virtual memory (UVM) [13], [43] or zero-copy access [31]. During iterative processing, memory slices containing active edges can be transferred to the

TABLE I: Runtime comparison of Subway and EMOGI on variable algorithms and datasets.

	SK-2005 graph		PageRank Algorithm	
	SSSP	**PageRank**	**sk-2005**	**uk-2007**
Subway	14.6(s)	**8.7**(s)	**8.7**(s)	16.9(s)
EMOGI	**7.5**(s)	18.6(s)	18.6(s)	**12.4**(s)

GPU implicitly in a user-transparent manner. Due to the fixed data migration mechanisms of ImpTM approaches, the transfer efficiency is highly sensitive to the graph's access pattern.

Having made extensive analysis, we find that a decision to choose one or the other approach for the best performance is determined by the memory access pattern of active edges. In GPU-accelerated graph processing frameworks based on a single approach, the performance is often suboptimal. Table I shows the performance comparison of Subway [38] (a ExpTM-compaction-based framework) and EMOGI [31] (an ImpTM-zero-copy-based framework). On sk-2005 graph [3], EMOGI outperforms the Subway on Single Source Shortest Path algorithm (SSSP) , but it losses on PageRank. In contrast, for the PageRank algorithm, Subway beats EMOGI on SK dataset [3], but losses on UK dataset [3]. Moreover, these approaches focus on reducing redundant active subgraph transmission and lack efficient mechanisms to reuse transferred graph data iterations. Existing frameworks [13], [40] based on page-centric data caching can hardly adapt to graph processing tasks with irregular and sparse memory access patterns.

We present HyTGraph, a GPU-accelerated Graph processing system that distinguishes itself from previous frameworks by not exclusively relying on either **ExpTM** or **ImpTM**. Instead, our system employs a **Hybrid Transfer Management** method (**HyTM**) that combines ExpTM and ImpTM to maximize performance. HyTM splits the graph into small partitions as ExpTM does. During iterative processing, HyTM estimates ExpTM cost and ImpTM cost on-the-fly by analyzing the edge access pattern of each partition, and chooses the most cost-efficient transfer method. Building upon **HyTM** method, HyTGraph provides an effective vertex-centric graph caching method incorporating fine-grained cache management and GPU parallel refreshing for efficient graph transfer reusing across iterations. Furthermore, HyTGraph provides a contribution-driven asynchronous scheduling to accelerate convergence. Experimental results on real-world and synthetic graphs demonstrate that HyTGraph achieves an average speedup of 5.0X over Subway [38], 2.0X over Grus [43], and 2.5X over EMOGI [31].

This work extends the conference version [44], enhancing the communication efficiency through fine-grained GPU graph caching: (1) We present a vertex-centric graph caching framework that reduces CPU-GPU communication through fine-grained GPU data caching. The framework further incorporates a decoupled cache refreshing mechanism along with GPU parallel processing to achieve low-overhead cache management (Section VII). (2) We conduct experimental studies to verify the effectiveness of the caching method in Section IX. These include: (a) Updates to the overall performance evaluation and discussion of HyTGraph in Sections IV and IX-G, incorporating the caching optimizations (b) Supplement data caching components in Sections IX-E and IX-F to assess the data transfer reduction and consequent performance improvement; (c) New experimental evaluations in Section IX-F to assess the cost and benefit of GPU data caching.

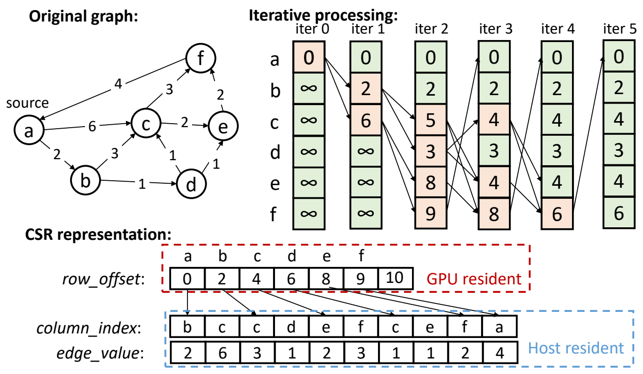

Fig. 1: An example of SSSP computation. The orange box indicates the active vertices and the green box indicates the inactive one. The input graph is organized into a CSR format.

II. Background

A. Vertex-Centric Graph Processing

Vertex-centric programming [16], [29] has been widely adopted in graph processing frameworks for its simplicity and powerful expression ability. It uses a generic function to define the behavior of a vertex and its neighbors. Considering the message passing direction, the function can be either pull- or push-based [42]. During computation, the function is iteratively evaluated on all vertices until the algorithm terminates. Figure 1 illustrates an example of SSSP, an algorithm to find the shortest paths from a given source vertex to all the other vertices. It starts from a source vertex a. In each iteration, the accessed vertices broadcast their shortest distances to the outgoing neighbors and update the shortest distance if the new path is shorter than the old one. The algorithm converges when no more vertices are updated. During iterative computation, only the vertices updated by the previous iteration (active vertices) need to be processed.

GPU graph processing. Recent research explores the massive parallelism of GPUs [23], [37], [42], [46], [51] to accelerate graph processing. Despite achieving promising results, the processing capability of these studies is limited by the GPU memory. For example, a common 16GB GPU can accommodate only about 600 million edges (assuming each edge occupies 8 bytes). While multi-GPU processing is an intuitive approach for scaling to large-scale graphs, the cost of expanding GPU memory is prohibitively high, with the price of 1GB of GPU memory often tens or even hundreds of times that of 1GB of CPU memory [33], [34]. Fortunately, we observe that vertex data, which requires frequent random accesses, is often exponentially smaller than edge data. Although edge data consumes significant memory, it is read-only and typically accessed sequentially. Therefore, placing vertex data on GPUs and offloading large-scale edge data to the CPU provides a cost-effective solution that leverages the high GPU computational power and the affordable host memory resources [31], [38]. A heterogeneous computing platform with a single 16GB GPU and 320GB of CPU memory can handle large-scale graphs with hundreds of millions of vertices and tens of billions of edges. In contrast, achieving this with multi-GPU in-memory processing could require dozens of such GPUs, along with the complexity of optimizing inter-GPU communication. If the input graph scales further, combining out-of-memory processing with multi-GPUs could provide a promising solution. This would involve partitioning vertex data across multiple GPUs and exploring new interconnect technologies to extend host memory, which is discussed as future work in Section X. In this work, we focus on optimizing the data communication in CPU-GPU heterogeneous computing, especially exploring efficient transfer management modules and caching mechanisms to minimize unnecessary edge communications.

B. ***ExpTM*** *Approaches*

ExpTM-filter. GraphReduce [39], GTS [24], and Graphie [18] adopt a filter-based method to reduce the inactive subgraph transfer. They monitor the active edges of the partitioned subgraphs and transfer only those containing active edges. Figure 2 (a) shows an illustrative example. This method filters out partitions containing no active edges without additional processing. Therefore, active partitions will be entirely transferred to the GPU, even if only one edge is active. When the proportion of active edges is low, the volume of unnecessary data transfer will be large.

ExpTM-compaction. In contrast, some other frameworks [38], [40], [50] introduce CPU-assisted compaction to reduce communication. Before transferring a partition containing active edges to the GPU, these frameworks use CPUs to remove the inactive subgraph and compact the remaining data into a continuous memory space to facilitate explicit memory copy. Figure 2 (b) shows an illustrative example of Subway [38], a typical ExpTM-compaction-based system. Compared with the filter-based frameworks [18], [39], compaction-based frameworks can minimize the data transfers by removing all inactive edges. But at the cost, it involves additional CPU and memory manipulation overhead.

C. ***ImpTM*** *Approaches*

ImpTM-unified-memory. Unified-virtual-memory (UVM) defines a managed memory space where both GPU and CPU share a single address space, maintaining a coherent memory image [13], [43]. Figure 2 (c) shows an illustrative example. During computation, memory pages (4KB in default) containing requested data are automatically migrated to GPUs. UVM provides page-centric GPU data caching, enabling subsequent accesses to the same memory page to be served directly from the GPU's global memory, thus avoiding additional data transfers. However, the "automated migration" cost is not free. Migrating new pages to the GPU memory triggers page-fault processing, which requires not only data transfer but also significant costs in Translation Lookaside Buffer (TLB) invalidation and page fault updating [31].

ImpTM-zero-copy. Zero-copy memory access offers a more lightweight approach. The method maps pinned CPU memory to GPU address spaces, enabling GPUs to directly access CPU memory through the Transaction Layer Packet (TLP) of PCIe [31]. Compared to UVM-based method, zero-copy access provides a finer granularity of CPU-GPU data access. As per the PCIe 3.0 specification, up to 256 outstanding memory requests can be processed concurrently by each TLP, with each request accommodating data payloads of 32, 64, 96, or 128 bytes [31], tailored to the size of the various adjacency lists. This capability allows zero-copy access to facilitate simultaneous, fine-grained access to the edge data. Moreover, zero-copy access incurs lower transfer overhead than UVM-based approaches as it eliminates the need for page-fault handling. As a sacrifice, zero-copy access cannot provide data-reusing functions. Repeat accesses to the same data will cause multiple separate CPU-GPU data transfers.

III. Analysis of existing approaches: A Motivating Study

In this section, we conduct an experimental analysis of the existing approaches using two graph algorithms with diverse memory access patterns: SSSP and PageRank.

A. *Analysis of ExpTM*

ExpTM-filter (ExpTM-F). As mentioned above, filter-based ExpTM has a large amount of redundant transfers even if the proportion of active edge is low. We run PageRank and SSSP on friendster-konect [2] graph to explore the redundant data transfer problem with the partition number set to 256. Figure 3 (a) shows the proportion curves of active edges and partitions containing active edges (active partitions). We can observe that the proportion of active partitions does not decrease immediately with the proportion of active edges. For SSSP and PageRank algorithms, the active edges account for only 28.3% and 12.3% of the total transfer volume. Although ExpTM-filter method shows ineffectiveness with a small number of active edges, it exhibits advantages when dealing with subgraphs containing a large proportion of active edges. This is because it can maximize the utilization of PCIe bandwidth through the `cudaMemcpy()` function.

ExpTM-compaction (ExpTM-C). The compaction-based ExpTM significantly reduces data communication by transferring compacted active data. However, this approach involves substantial overhead due to CPU-based compaction processing, particularly when a large proportion of active edges are involved. As highlighted by Subway [38], in scenarios where the proportion of active edges is high (e.g., 80%), the overhead associated with compaction can even outweigh the benefit of transfer reduction [38]. Figure 3 (b) depicts the per-iteration runtime breakdown of Subway and showcases instances where costs exceed benefits. Additionally, Figure 3 (c) shows the overall performance breakdown of SSSP algorithm on Subway, with the preprocessing overhead excluded from the execution time. We can observe that across all five datasets, the compaction stage accounts for 34.5% of the overall runtime.

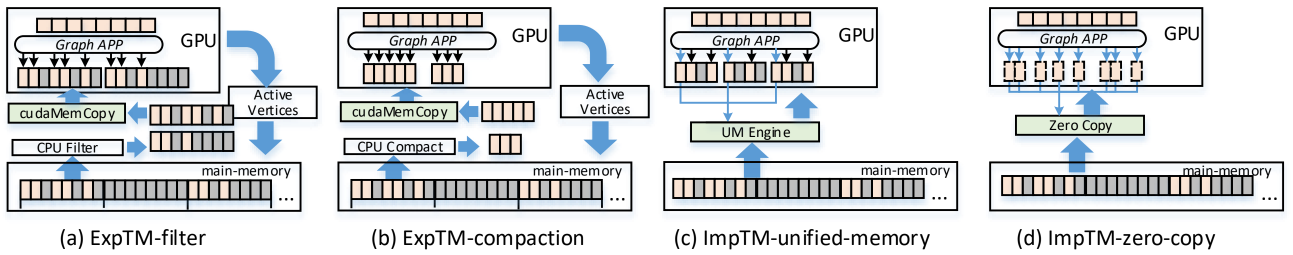

Fig. 2: An example of the four approaches. The thin blue arrow, thin black arrow, and thick blue arrow represent the remote memory access, local memory access, and host-GPU data transfer, respectively.ExpTM-based approaches need to transfer the active vertices back to the host side for compaction or filtering.

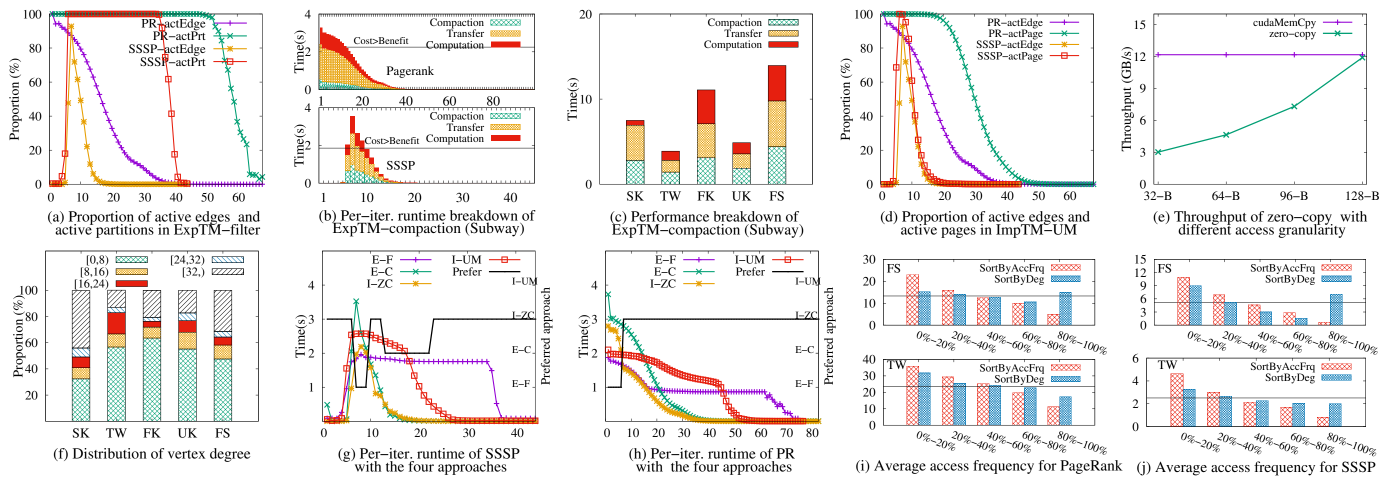

Fig. 3: Performance analysis of the four engines using the two algorithms.

B. Analysis of ImpTM

ImpTM-unified-memory (ImpTM-UM). Unified-virtual-memory provides on-demand memory migration. However, this method falls short of efficiency in graph processing applications. On the one hand, recent studies show that the peak bandwidth of unified-memory can only reach 73.9% of that of explicit memory copy due to the high “automated migration” overhead [31]. On the other hand, the migration granularity of 4K bytes often leads to the inclusion of substantial inactive data [13], [31] in each memory page. This issue is caused by the inherent irregularity and poor locality of graph algorithms [31], [40]. Figure 3 (d) shows the proportion of active edges and active memory pages of each iteration. For SSSP and PageRank algorithms, the active edges account for only 54.5% and 65.0% of the total transfer volume, respectively. Therefore, ImpTM-unified-memory exhibits suboptimal communication performance on large graphs, regardless of whether the ratio of active edges is high or low. Additionally, the page-centric data caching also contributes little to the performance improvement due to the low space utilization of active subgraphs in the memory pages. The UVM-based approach only shows efficiency when the graph size is sufficiently small to be fully accommodated within the GPU memory.

ImpTM-zero-copy (ImpTM-ZC). Maximizing PCIe bandwidth utilization is crucial for improving zero-copy access performance. As indicated by EMOGI [31], saturating most of the 256 memory requests in each TLP with 128-byte data is essential for maximizing the PCIe bandwidth. The underlying reason is as follows: Each TLP not only carries the payloads of memory requests but also includes a header field containing control information. Smaller memory requests result in a higher number of TLPs for the same data volume, thus allocating a significant portion of the bandwidth to header field transfers. Figure 3 (e) shows the throughput of zero-copy access across various memory request sizes (ranging from 32 bytes to 128 bytes). We can observe that when the memory request size is 128 bytes, the zero-copy access can achieve almost the same performance as `cudaMemcpy` (the maximum PCIe utilization). However, when the request size is set to 32 bytes, the throughput decreases significantly. To optimize bandwidth usage, EMOGI [31] proposes merged and aligned optimization with which each warp of threads access consecutive neighbors of one vertex in a 128-byte cache line size. Nonetheless, guaranteeing that most memory requests reach the 128-byte mark is challenging, especially when considering that each neighbor typically requires 4 bytes, and real-world graphs, adhering to a power-law distribution, often feature vertices with fewer than 32 neighbors. As shown in Figure 3 (f), on average, 74.7% of vertices have fewer than 32 neighbors, with 51.1% having fewer than 8. This leads to inconsistent performance of zero-copy access across real-world graphs.

C. Performance Comparison of the Four Approaches

We report the per-iteration runtime of ExpTM-filter, ExpTM-compaction, ImpTM-unified-memory, and ImpTM-zero-copy on friendster-konect [2] using two typical graph algorithms: the graph traversal algorithm SSSP and the iterative algorithm PageRank [42]), illustrated in Figure 3 (g) and (h). The implementations of ExpTM-filter, ImpTM-unified-memory, and ImpTM-zero-copy leverage the processing kernel of SEP-Graph [42]. The `cudaMemAdviseSetReadMostly` optimization is enabled for ImpTM-unified-memory (the evicted memory pages will be discarded directly instead of written back to the CPU memory). For the ExpTM-compaction implementation, We use Subway [38] due to its highly-optimized compaction engine and efficient GPU kernel from Tigr [37]. All methods are set to execute synchronously to maintain a consistent number of active vertices across iterations. We employ a "**Prefer**" curve to indicate the best-performing approach in each iteration. By examining the proportion curves of active edges for SSSP and PageRank in Figure 3 (a), we observe that ExpTM-filter excels when the proportion of active edges is high, attributable to its efficient utilization of PCIe bandwidth.

When the proportion of active edge is low, ImpTM-zero-copy outperforms other approaches in most iterations, thanks to its fine-grained data migration mechanism. However, for the SSSP algorithm, ExpTM-compaction occasionally outperforms ImpTM-zero-copy during certain iterations. This variability can be attributed to the unstable performance of zero-copy under different vertex degrees. As mentioned above, zero-copy's effectiveness is affected not only by the active edge proportion but also by the average degree. Given a fixed number of active edges, a higher count of active vertices leads to a higher number of fragmented edge data accesses and an increase in partially filled TLP requests, leading to reduced bandwidth utilization efficiency. ImpTM-unified-memory consistently underperforms across all scenarios, even with GPU data caching. Its coarse-grained page migration mechanism falls short of recognizing the irregular and sparse data access patterns of graph processing, thereby diminishing both cache and PCIe bandwidth efficiency.

D. Cross-iteration Data Reusing

Existing transfer management approaches primarily focus on intra-iteration communication optimization. However, duplicated vertex accesses across iterations also result in certain edge data being transferred multiple times. As shown in Figure 3 (i-j), we evaluate the SSSP and PageRank algorithms on the TW and FS graphs. Vertices are sorted and grouped into five categories based on their access frequency (red bars) and vertex degree (blue bars), and the average access frequency for each group is presented. The results reveal that the top 20% most frequently accessed vertices exhibit significantly higher access frequencies than the average (black line) and other groups, offering considerable opportunities for communication reuse. Existing data caching mechanisms, however, exhibit certain limitations. The ImpTM-unified-memory approach, which employs page-centric dynamic data migration and caching, fails to adapt effectively to the irregular access patterns of graph data. Recent studies have explored the significance of high-degree vertices (i.e., hub vertices) in graph processing, highlighting their frequent access during computation [27]. Consequently, extracting hub vertices for data caching becomes a natural and intuitive solution. However, we observe

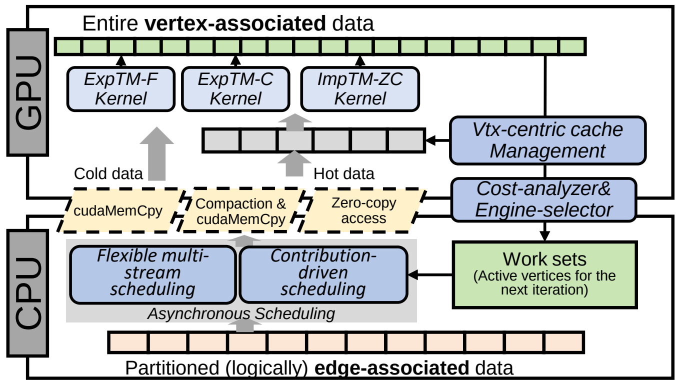
GPU
CPU
Entire **vertex-associated** data
ExpTM-F Kernel
ExpTM-C Kernel
ImpTM-ZC Kernel
Vtx-centric cache Management
Cold data
Hot data
cudaMemCpy
Compaction & cudaMemCpy
Zero-copy access
Cost-analyzer& Engine-selector
Flexible multi-stream scheduling
Contribution-driven scheduling
Asynchronous Scheduling
Work sets (Active vertices for the next iteration)
Partitioned (logically) **edge-associated** data

Fig. 4: Overview of HyTGraph.

that high-degree vertices in different graph processing tasks do not always align with high-frequency accessed vertices. as shown by the blue columns in Figure 3 (i-j). On the TW dataset, the bottom 20% low-degree vertices exhibit even higher access frequencies than the middle groups. Moreover, the overlap between the top 20% of vertices ranked by degree and those ranked by access frequency is limited, ranging from 34% to 45%. This inspires us to design a more targeted, access frequency-guided caching mechanism for efficient data reuse.

E. Summary of Existing Approaches

Having made intensive analysis, we observe that the data transfer overhead is affected by three primary factors: data transfer volume, PCIe bandwidth Utilization, and CPU pruning cost. As depicted in Table II, existing approaches fail to optimize these factors cooperatively. Adopting a single transfer management method often limits the system's effectiveness to only one or several specific scenarios, as shown in the last column in Table II. Additionally, existing GPU data caching mechanisms, which rely on fixed-size memory page migration or degree-based subgraph pre-selection, are not suited for handling the irregular data access patterns inherent in graph processing. In addition to systems [13], [31], [38], [39] mentioned above, Scaph [50] adopt ExpTM-compaction. Different from Subway, Scaph [50] and Ascetic perform compaction on the partitioned graph. It distinguishes the partitions with a small proportion of active edges and compacts them for subsequent GPU processing. In contrast, partitions with a large proportion of active edges are entirely loaded to the GPU. Grus [43] is an ImpTM-based framework that manages the edge-associated data in main memory with priorities, prefetching high-priority data to the GPU through unified-memory and accessing low-priority data through zero-copy. In addition, some frameworks [14], [28] also use CPU-GPU co-processing to accelerate graph processing. We review them in Section XI.

IV. HyTGraph Overview

We present HyTGraph, a GPU-accelerated graph processing system based on hybrid transfer management (HyTM) to maximize performance. Figure 4 shows an overview. HyTGraph organizes the graph into a CSR structure. Following [18], [50], HyTGraph logically splits the edge data into N edge-balanced partitions $\{P_0, P_1 \ldots, P_{N-1}\}$. During computation,

TABLE II: Summary of existing approaches and representative systems.

Approach	System	Transfer Volume	Bandwidth Utilization	Prune Cost	GPU Data Cache	Preferred Scenario
ExpTM-F	GraphReduce [39] Graphie [18] GTS [24]	High	High	Low	N/A	•Subgraph with a large proportion of active edges
ExpTM-C	Subway [38] Scaph [50] Ascetic [40]	Low	High	High	N/A	•Subgraph with a small proportion of active edges and small average degree
ImpTM-UM	HALO [13] Grus [43]	Medium	Low	NA	Page-centric	• Small graph that can fit into GPU memory
ImpTM-ZC	EMOGI [31]	Low	Unstable	No	N/A	• Subgraph with a small proportion of active edges and high average degree
HyTM	Our approach	Low	high	Low	Vertex-centric	• Adapt to subgraph with various active degree

partitions containing active vertices are processed with their most cost-efficient engines. HyTGraph provides three functions to achieve efficient HyTM.

Cost-based engine selection. HyTGraph uses a GPU-resides cost analyzer and engine selector to compute the data transfer overhead for different approaches and selects the most cost-efficient one for each partition. Based on the analysis in section III, we choose ExpTM-F, ExpTM-C, and ImpTM-ZC as engine candidates. In addition, HyTGraph provides a task combining optimization to merge small subgraph pieces into larger tasks to minimize scheduling overhead.

Contribution-driven asynchronous scheduling. HyTGraph introduces asynchrony to improve computation efficiency. Rather than simply recomputing the loaded subgraph multiple times [38], [50], HyTGraph adopts a contribution-driven priority scheduling method to prioritize those partitions that contribute more to convergence. To further improve resource utilization, HyTGraph uses multiple CUDA streams to overlap the computation kernel, data transfer, and CPU-based active subgraph compaction.

Vertex-centric graph cache. HyTGraph provides a vertex-centric graph caching method, distinguishing itself from the page-centric methods in existing GPU graph processing systems [13], [40]. HyTGraph finely caches the frequently accessed data and compactly stores them in the GPU to maximize the cache utilization. Moreover, instead of performing the heavy-weight individual-vertex data replacement, HyTGraph uses a batched cache refreshing method that minimizes data replacement overhead through GPU parallel processing.

V. Cost-based Engine Selection

A. Cost Analysis and Engine Selection

Most existing activeness-tracking-based frameworks use the proportion of active edges as the evaluation metric [18], [28], [39], [50] to select appropriate processing engines, which provides an intuitive and lightweight distinguishing method. However, it is not suitable in HyTM as the proportion of active edges cannot reflect the cost of different transfer approaches (as detailed in III-C). In this work, we directly model the overhead of different transfer management methods and accurately select the most cost-efficient execution engine.

Cost of ExpTM-filter. The ExpTM-filter-based approach entirely transfers partitions containing active edges to the GPU using `cudaMemcpy`, thus incurring only data transfer overhead. This overhead can be estimated by the saturated TLPs (as discussed in Section III, Figure 3 (e)). Given a partition P_i, the number of memory requests can be calculated with $\sum_{v \in P_i} D_o(v) * d_1/m$, where $\sum_{v \in P_i} D_o(v)$ is the edge number of partition i, $D_o(v)$ represents the number of neighbors of v, d_1 represents the memory occupation of a vertex, and m represents the maximum capacity of an outstanding memory request (128-bytes in PCIe-3.0 specification). Denote MR as the maximum number of outstanding memory requests in TLP ($MR = 256$ in PCIe 3.0) and $\lceil \cdot \rceil$ as the round-up operation, we formalize the transfer overhead of each P_i as follow:

$$Tef_i = \Big\lceil \big(\sum_{v \in P_i} D_o(v) \big) * d_1/m/MR \Big\rceil * RTT, \qquad (1)$$

where $\Big\lceil \big(\sum_{v \in P_i} D_o(v) \big) * d1/m/MR \Big\rceil$ is the number of required TLPs, and RTT represents the round trip time for PCIe to process a saturated TLP request.

Cost of ExpTM-compaction. ExpTM-compaction involves additional CPU-based compaction, so its cost consists of two parts: the data transfer overhead and the CPU compaction overhead. Since the compaction needs to reorganize the active edges and change their positions, we also need to generate a vertex index array and transfer it to GPU for addressing the compacted neighbors. Then the transfer volume can be formalized as $\sum_{v \in A_i} D_o(v) * d_1 + |A_i| * d_2$, where A_i represents the active vertex subset of P_i and d_2 represents the memory occupation of each vertex index. The CPU-based compaction is related to both data transfer volume and CPU compaction throughput. which can be computed using $\sum_{v \in A_i} D_o(v) * d_1 + |A_i| * d_2 / Thpt_{cpt}$, where $Thpt_{cpt}$ represents the CPU compaction throughput. Then, the cost of ExpTM-compaction can be formalized as:

$$\begin{aligned} Tec_i = & \Big\lceil \big(\sum_{v \in A_i} D_o(v) * d_1 + |A_i| * d_2 \big)/m/MR \Big\rceil * RTT \\ & + \sum_{v \in A_i} D_o(v) * d_1 + |A_i| * d_2 / Thpt_{cpt} \end{aligned} \qquad (2)$$

Cost of ImpTM-zero-copy. The ImpTM-zero-copy approach provides vertex-centric on-demand access in the cacheline size. Therefore, each active vertex v takes one or several independent memory requests depending on its neighbor size. Generally, the number of required memory requests of vertex v can be formalized as $\lceil D_o(v) * d_1/m \rceil$. Considering that we can hardly guarantee the neighbors of all vertices start from a cacheline-aligned position, some vertices may have the misaligned neighbor array and thus require one additional memory transaction [31]. We introduce a function $am()$, which returns 1 for unaligned vertices and 0 for the others[1]. Therefore, the transfer overhead of ImpTM-zero-copy can be formalized as:

$$Tiz_i = \Big\lceil \Big(\sum_{v \in A_i} \big(\lceil D_o(v) * d_1/m \rceil + am(v) \big) \Big) / MR \Big\rceil * RTT_{zc}, \tag{3}$$

where $\Big(\sum_{v \in P_i(V) \cap A_i} \lceil D_o(v) * d_1/m \rceil + am(v) \Big)$ is the required memory transactions of P_i. It should be noted that the TLP round trip time of zero-copy (RTT_{zc}) is not the same as that in ExpTM (RTT) because the payload of each TLP in zero-copy may be unsaturated. This makes RTT_{zc} always less than the RTTs in ExpTM-filter and ExpTM-compaction. In this paper, we use a dumpling factor γ to compute RTT_{zc} for each partition as follows: $RTT_{zc} = \gamma * RTT + (1-\gamma) * (\sum_{v \in A_i} D_o(v) / \sum_{v \in P_i} D_o(v)) * RTT$, where $(\sum_{v \in A_i} D_o(v) / \sum_{v \in P_i} D_o(v)$ is the proportion of active edge. $\gamma * RTT$ represents the fixed time to process a TLP, and $(1-\gamma) * (\sum_{v \in A_i} D_o(v) / \sum_{v \in P_i} D_o(v)) * RTT$ represents the time related to the size of payload. By referring to [31], we set γ to 0.625.

Transfer engine selection. HyTGraph compares Tef_i, Tec_i, and Tiz_i to choose the most cost-efficient execution engine. However, theoretically modeling the CPU compaction throughput $Thpt_{cpt}$ in Equation 2 is challenging because ExpTM-compaction introduces parallel and random writes on the host memory. This makes $Thpt_{cpt}$ vary with active edges nonlinearly. In practice, we compute Tec_i by considering only the transfer overhead and compare it with Tef_i and Tiz_i. If Tec_i is less than $\alpha * Tef_i$ and Tec_i is less than $\beta * Tiz_i$, we choose ExpTM-compaction. The first condition comes from Subway's observation [38], where α is set to 80%. The second condition is based on the observation from Section III: when a partitioned subgraph has few active edges but many active vertices, and zero-copy requires many unsaturated memory requests to transfer the data, leading to ExpTM-compaction is a better choice than ImpTM-zero-copy. In the implementation, β is set to 40%. If these conditions are not met, we compare Tiz_i with Tef_i. If Tiz_i is less than Tef_i, we choose ImpTM-zero-copy. Otherwise, we choose ExpTM-filter. The value of RTT can be arbitrarily specified, as it can eliminated in subsequent comparisons. Since the cost computation between partitions is independent, HyTGraph performs the engine selection on the GPU for high performance. Algorithm 1 line (2-13) shows the overall execution flow.

B. Partition Granularity and Task Combination

A key to implementing high-performance hybrid transfer management is to determine the optimal granularity for task scheduling. The existing frameworks [18], [28], [39], [50] directly use the partitioned subgraphs as scheduling unit. This method is simple and straightforward, however, can lead to low task scheduling performance. If the partition size is too large, the coarse-grained cost computation may lead to inappropriate engine selection. If the partition size is small, it may lead to higher kernel scheduling overhead and fragmented data transfers for more partitions.

[1]In the implementation, the number of memory requests of each active vertex $\lceil D_o(v) * d_1/m \rceil + am(v)$ can be directly computed by using the length and physical position of the neighbor data.

Algorithm 1 Cost-based engine selection

```
Input: active vertex set $\{A_0, \cdots, A_{N-1}\}$ of $N$ partitions,
Output: tasks prefer ExpTM-filter $\{Vf_0 \dots Vf_{M-1}\}$ ($M < N$), task
    prefer ExpTM-compaction $Vc$, and task prefer ImpTM-zero-copy $Vz$.
 1: initialize a selection array $\{p_0, \dots p_{N-1}\}$ on GPU.
    Cost analysis and engine selection:
 2: for each $A_i$ in $\{A_0, \cdots, A_{N-1}\}$ do in parallel
 3:     Compute $Tef_i$, $Tec_i$, and $Tiz_i$ according to Formula (1,2,3)
 4:     if $Tec_i < \alpha * Tef_i$ and $Tec_i < \beta * Tiz_i$ then
 5:         $p_i$='ExpTM-C';
 6:         insert $A_i$ to $Vc$; //pre-combine on GPU
 7:     else if $Tef_i < Tiz_i$ then
 8:         $p_i$='ExpTM-F';
 9:     else
10:         $p_i$='ImpTM-ZC';
11:         insert $A_i$ to $Vz$; //pre-combine on GPU
12:     end if
13: end for
14: Copy $Vc$, $\{p_0, \dots p_{N-1}\}$ and $\{A_0, \cdots, A_{N-1}\}$ to host.
    Task Combination:
15: $i = 0$, $j = 0$, $length = 0$;
16: while $i < N$ do
17:     if $p_i$=='ExpTM-F' and $length < k$ then
18:         insert $A_i$ to $Vf_j$;
19:         $length = length + 1$;
20:     else
21:         $length = 0, j = j + 1$;
22:     end if
23:     $i = i + 1$;
24: end while
```

To achieve fine-grained engine selection and low overhead task scheduling simultaneously, HyTGraph decouples the graph partitioning and task scheduling and optimizes them separately. HyTGraph partitions the graph into small partitions (32MB each partition) to provide fine-grained cost analysis. While in the computation, HyTGraph packages partitions choosing the same engine into large chunks for processing. Specifically, for partitions using ExpTM-filter, HyTGraph merges k consecutive partitions into a large one (k=4 in HyTGraph) to reduce scheduling overhead (Line 15-24 in algorithm 1). For partitions using ExpTM-compaction, HyTGraph merges all their active vertices and writes their neighbors to a consecutive memory chunk to leverage explicit memory copy (line 6 in algorithm 1). For partitions using ImpTM-zero-copy, HyTGraph merges all active vertices (line 11 in algorithm 1) and processes them with one CUDA kernel to leverage the implicit computation-communication overlapping feature of zero-copy access.

VI. Contribution-Driven Asynchronous Task Scheduling

Asynchronous computation allows newly updated results to be used immediately in subsequent computation, which has been proven to be effective in GPU-based graph processing [6], [42]. However, simply processing the transferred subgraph multiple times may cause the local updates to be abolished by the subsequent computation from other partitions, leading to increased computation and data transfer. This is known as the stale computation problem [11], [45]. Our experiments reveal that existing framework with multi-round processing can even cause higher transfer volume (See Section IX-E for details) than synchronous scheduling in some cases. To address this issue, HyTGraph adopts a contribution-driven priority scheduling method.

Hub-vertex-driven priority scheduling. Due to the power-law property of real-world graphs, some important vertices with high incoming and outgoing degrees often become the hubs in the computation path. These vertices become critical upstream dependencies of a large number of vertices because of the large outgoing degree. On the other hand, these vertices also have a high probability of being activated in the iterative computation due to large incoming degrees. If these vertices do not accumulate sufficient updates before being scheduled, the downstream computation results based on the current value are likely to be abolished by subsequent new updates. Based on this observation, we propose a hub-vertex-driven priority scheduling approach. By ensuring that the hub vertices accumulate enough contributions before being scheduled, HyTGraph can reduce the possible stale computations on the downstream vertices. Implementing hub-vertex-driven scheduling in GPU-accelerated platforms is challenging, as the hub vertices distribute randomly among the entire graph, which makes hub vertices hard to gather and transfer. To solve this problem, HyTGraph adopts a hub vertex sorting method [48], that groups the top 8% important vertices at the beginning of the CSR structure. The importance score of each vertex is measured by the following equation:

$$H(v) = \frac{D_o(v) * D_i(v)}{D_{omax} * D_{imax}} \tag{4}$$

$D_i(v)$, $D_o(v)$, D_{imax}, and D_{omax} represent the incoming-, outgoing-, maximum incoming-, and maximum outgoing-degree, respectively. In this way, the hub vertices are grouped together, and the non-hub-vertices remain in their natural order. HyTGraph recomputes the loaded subgraph only once because most updates can only pass two hops effectively [41]. Another benefit of this method is that the vertices having a high probability of being activated (with large in-degree) are stored together, which can help gather high-active subgraphs for the cost-based engine selection. It is worth mentioning that the hub sorting does not need to be performed every time. The hub vertex can be sorted only once in the preparation stage, and all subsequent executions (of different algorithms) can benefit from it.

Δ-driven priority scheduling. For iterative graph algorithms based on arithmetic accumulation, e.g., Δ-based PageRank and PHP [47], the contribution of vertices is directly reflected in their delta values (the messages to be accumulated). Prioritizing vertices with larger Δ value can help downstream vertices accumulate more valid updates [47]. Following the original vertex-centric Δ-driven priority scheduling, HyTGraph provides a partition-centric Δ-driven scheduling method that computes an overall Δ value for each partition and prioritizes those with larger contributions.

Algorithm 2 Data caching during iterative processing.

```
1: $V_{curr}$=init("ALG") // initial active vertices
2: $V_{cache} = \emptyset$ // cached vertices
3: $N_{batch} = 0$ // counter of processed vertices
4: while $V_{curr} \neq \emptyset$ do
5:     if $N_{batch} > \gamma|V|$ then
6:         $V_{cand}$=hot_candidates_VCDC($C_{cap}$)
7:         if overlap($V_{curr}$,$V_{cache}$) then
8:             cache_update_VCDC($V_{curr}$, $V_{cache}$)
9:                 Evict cold data and alloc space for new candidates
10:            $c\_tag = 1$
11:        end if
12:        reset_hotness_VCDC()
13:        $N_{batch} = 0$
14:    end if
15:    $\{V_c, V_z, V_f\}$=engine_selection_HyTM($V_{curr}$)
16:    $V_{next}$=proc_HyTM_VCGC($\{V_c, V_z, V_f\}$,$V_{cand} \setminus V_{cache}$)
17:        Load accessed candidates into the cache
18:        Record the number of accesses
19:    $V_{curr} = V_{next}$
20:    $N_{batch} += |V_{curr}|$
21:    if $c\_tag == 1$ then
22:        $V_{cache} = V_{cand}$
23:    end if
24: end while
```

VII. Vertex-Centric Graph Caching

The proposed hybrid transfer management focuses on transfer reduction within each iteration. However, certain iterative graph algorithms (e.g., Pagerank) involve repeated data access among iterations [40], which benefit less from HyTM, and can hardly optimized with exiting caching mechanisms due to coarse-grained data caching and challenging cache vertices determination. In this section, we introduce an access frequency-guided, vertex-centric graph caching (VCGC) method. During computation, HyTGraph tracks the access frequency of all vertices in real time, sorts them in by access frequency to determine the cache candidates, and performs vertex-centric cache replacement using the hybrid communication engine. This approach leverages a more directed access frequency metric and finely transfers and caches the graph in a vertex-centric manner.

Enable high-performance VCGC is a non-trivial task as frequent eviction and loading for variable-length adjacency lists on the GPU incur substantial memory manipulation overhead. To address this issue, HyTGraph provides several key features, including 1) a periodic cache refreshing approach that decouples the eviction and loading of frequently accessed data; 2) CSR-based compacted data organization to maximize memory utilization, and uses GPU parallel processing to accelerate the cache determination. We first introduce the overall execution flow and then detail the design and implementation of each component.

Execution flow. The cache management flow during iterative processing is outlined in Algorithm 2, with functions associated with data caching indicated by the **VCDC** suffix. To begin, HyTGraph initializes the active vertices (Line 1) and the necessary data for cache maintenance (Line 2 and 3) and then performs the iterative computation (Line 4-24). Before engine selection and computation, HyTGraph first checks whether the cache condition is triggered based on the hotness (Line 5-7). If the condition is met, HyTGraph refreshes the vertex-centric cache by evicting cold data and reloading the new data in batches through GPU parallel processing. To avoid additional host-GPU data transfer, HyTGraph separates the cache filling and eviction, first evicting cold data and allocating space for the new candidates (in Line 8) and then loading the data to the GPU during HyTM processing. This design ensures low cache management overhead.

Hotness-based cache candidate selection. HyTGraph determines the cache candidates by sorting vertices by the hotness and obtaining the top-K vertices as the candidates. The "hotness" of a vertex is defined by the access volume over a certain period, which is maintained using a $|V|$-length hotness array during iterative processing. In the candidate determination stage (Line 8), vertices are sorted by the hotness, with the topK hot vertices whose aggregated neighbor sizes do not exceed a given capacity (C_{cap}) as the candidates, i.e., V_{cand}. Following the selection, the hotness array is reset for the next step. This process, encompassing hotness evaluation and top-K computation, incurs less overhead compared to iterative processing. HyTGraph also performs the selection process on the GPU to achieve high performance.

Cache refresh condition checking. Determining appropriate replacement timing is crucial to efficient communication and low replacement overhead. Length replacement cause increased cache miss, while highly frequent replacements can result in the overhead outweighing the benefits. To address this issue, HyTGraph employs a lazy condition-checking approach to balance the cost and benefit. Firstly, instead of using iterations as the metric for timing checks, HyTGraph calculates the cache candidates V_{cand} after processing a given number of vertices, denoted by $\gamma|V|$ (Line 5). This is to adapt to the asynchronous and Δ-based priority processing in HyTGraph. Secondly, HyTGraph evaluates whether the data requiring replacement exceeds β of the new candidates (Line 7) and performs cache refreshing (Line 8) only if the condition is met. Otherwise, iterative computation continues with the old cache. The 30% threshold is configured based on the insight that it ensures sufficient changes in access frequency are accumulated while avoiding frequent cache refreshing caused by minor fluctuations in vertex hotness. In graph computation, the access frequency distribution typically stabilizes after the first few iterations. Therefore, this 30% setting allows HyTGraph to cache most of the correct vertices after a few refresh cycles. Once stabilized, the system avoids unnecessary adjustments triggered by small changes, which would otherwise require reorganizing the entire CSR structure.

Vertex-centric cache refreshing. Cache replacement involves the removal of old vertices and the loading of new ones. HyTGraph decouples them into two phases and processes them separately to reduce replacement overhead. In the cache updating stage (Line 8), HyTGraph performs GPU parallel cold data deletion and compacts the remaining data to make room for the new candidates (Line 8). Initially, a data deletion kernel is launched to identify all outdated vertices ($V_{cache} \setminus V_{cand}$) and mark them as invalid. Subsequently, the remaining valid data ($V_{cand} \cap V_{cache}$) is compacted in the cache. Finally, new spaces are allocated for the new vertices ($V_{cand} \setminus V_{cache}$) in the tail of the cache, and the corresponding write indices are generated. To avoid additional data communication for loading data into the cache, HyTGraph integrates the cache loading stage into the computation kernel (Line 16). During computation, the GPU computation kernel will store the accessed data of new cache candidates ($V_{cand} \setminus V_{cache}$) in the allocated cache space according to the indices generated in the cache updating stage. Such an implementation ensures cache refreshing efficiency by fully utilizing GPU parallelism and avoiding additional host-GPU data communications.

VIII. Implementation and Optimizations

Flexible multi-stream scheduling. The processing engines of ExpTM-F, ExpTM-C, and ImpTM-zero-copy-ZC require different resources, including CPUs for active edge compaction, GPU for the computation kernel, and PCIe for the host-GPU data transfer. To overlap the resource utilization and improve the parallelism, HyTGraph uses multiple CUDA streams to process the tasks concurrently. During the iterative processing, the task scheduler monitors the available streams and assigns them to tasks that have not been scheduled. The operating system will automatically overlap data transfer and kernel computation of different streams. HyTGraph first schedules the ExpTM-Filter tasks with specific priority (as discussed in Section VI) to leverage the contribution-driven priority scheduling. Then the ImpTM-zero-copy and ExpTM-compaction tasks are scheduled. The CPU-based active edge compaction can be overlapped with the kernel computation and data transfer of ImpTM-zero-copy and ExpTM-filter.

Hotness computation. HyTGraph uses a byte per vertex to record the access frequency during computation, optimizing GPU memory usage. The involved sorting, TopK, and compaction operations are implemented using the CUB library [8].

CPU data compaction. HyTGraph provides a simple yet efficient compaction engine on the CPU following the deign of Subway [38]. To track locations of compacted edge data, HyTGraph re-generate a new compressed neighbor index for fast edge location.

Computation kernel. HyTGraph uses SEP-Graph's processing kernel for its mature optimizations and enables neighbor shifting [42] for it to support ExpTM-F and ExpTM-C engines.

IX. Experimental Evaluation

A. Experimental Setup

Environments. Our main test platform is equipped with one Intel Silver 4210 2.20Ghz 10-core CPU, 128GB DRAM, and an NVIDIA GTX 2080Ti GPU with 34SMX clusters, 4352 cores, and 11GB GDDR6 memory. The GPU is enabled with CUDA 10.1 runtime and 418.67 driver. The host side runs Ubuntu 18.04 with Linux kernel version 4.13.0. All source codes are compiled with `-O3` optimization.

Graph algorithms and datasets. We evaluate HyTGraph with four algorithms. Besides SSSP and PageRank, the other two algorithms are Breadth-First Search (BFS) and Connect Component (CC) [42]. We use both real-world graphs and

TABLE III: Dataset description.

Dataset	|V|	|E|	|E|/|V|	Size
Road-USA [2] (RU)	23.9M	57.7M	2.4	461M
Orkut [2] (RU)	3.1M	117M	39	968M
sk-2005 [3] (SK)	50.6M	1.93B	38	28GB
Twitter [2] (TW)	52.5M	1.96B	37	32GB
Friendster-konect [2] (FK)	68.3M	2.59B	37	42GB
uk-2007 [3] (UK)	105.1M	3.31B	31	55GB
Friendster-snap [2] (FS)	65.6M	3.61B	55	58GB
RMAT [22]	1-100M	0.1-6.4B	-	-

synthesized graphs in our evaluation. The major parameters of graph datasets that are used in our experiments are presented in Table III. The synthetic graphs are generated by PaRMAT [22] with the input parameters $a = 0.5$, $b = 0.2$, and $c = 0.2$ to maintain a power-law distribution.

Systems for comparison. We compare HyTGraph with three representative GPU-accelerated graph processing systems Subway [38], EMOGI [31], and Grus [43]; a in-memory GPU graph processing system cuGraph [9]; and a CPU-based graph processing system Galois [32]. Additionally, we also implement ExpTM-filter and ImpTM-unified-memory in HyTGraph's codebase for a fair comparison. We use the default configuration of these systems. The number of compaction threads in Subway and HyTGraph is set to 2×CPU cores. The cache capacity of HyTGraph is set to 2GB, which is the available memory when processing the largest graph and aligned for all datasets. All reported results are measured by averaging the number of 5 runs. HyTGraph is open-sourced at https://github.com/iDC-NEU/HyTGraph.

TABLE IV: Comparison with other systems.

		Overall runtime (s)						
Alg.	System	RU	OK	SK	TW	FK	UK	FS
PR	Galois	4.68	4.66	21.3	66.3	293.6	28.5	342.4
	cuGraph	0.31	0.5	-	-	-	-	-
	ExpTM-F	6.41	3.54	37.7	34.8	60.7	34.3	162.8
	ImpTM-UM	0.76	0.22	6.89	16.5	75.4	22.4	102.7
	Grus	0.60	**0.17**	**1.72**	12.2	52.2	14.8	79.8
	Subway	**0.17**	0.26	8.68	38.1	73.7	16.9	108.4
	EMOGI	3.11	0.52	18.6	21.4	51.1	12.4	68.3
	HyTGraph	1.67	0.34	1.84	**8.76**	**21.6**	**4.04**	**29.2**
SSSP	Galois	23.92	1.71	26.7	12.9	51.5	15.2	33.1
	cuGraph	8.05	0.89	-	-	-	-	-
	ExpTM-F	68.5	2.14	60.9	15.1	50.4	60.9	70.1
	ImpTM-UM	3.57	**0.18**	12.7	10.1	37.2	18.6	34.9
	Grus	14.98	0.20	25.2	11.2	70.8	5.32	16.9
	Subway	**2.65**	0.21	14.6	10.9	20.8	18.4	27.7
	EMOGI	33.75	0.38	7.46	4.09	14.9	4.71	11.8
	HyTGraph	5.24	0.24	**5.02**	**1.81**	**7.08**	**2.56**	**5.86**
CC	Galois	20.8	0.41	23.9	15.7	35.9	55.1	39.4
	cuGraph	**0.31**	0.22	-	-	-	-	-
	ExpTM-F	36.21	0.15	21.9	5.47	10.9	41.6	11.8
	ImpTM-UM	3.25	0.15	**1.43**	1.49	3.27	7.88	4.16
	Grus	12.37	**0.13**	2.09	1.36	3.21	5.17	4.69
	Subway	2.99	0.19	11.67	6.52	8.61	14.7	14.1
	EMOGI	14.95	0.21	4.01	1.96	2.71	4.54	3.76
	HyTGraph	5.25	0.18	3.23	**1.19**	**2.07**	**3.23**	**2.56**
BFS	Galois	**1.28**	0.88	16.2	7.55	12.5	15.2	14.7
	cuGraph	1.66	0.68	-	-	-	-	-
	ExpTM-F	26.87	1.02	20.3	3.86	8.87	25.1	9.54
	ImpTM-UM	2.45	0.45	1.13	1.29	1.97	2.33	6.25
	Grus	9.90	**0.13**	**0.83**	1.11	1.85	2.37	3.35
	Subway	1.63	0.28	7.39	5.79	6.85	9.04	13.49
	EMOGI	9.93	0.24	1.06	1.04	**1.44**	1.26	**1.97**
	HyTGraph	3.75	0.22	0.93	**0.85**	1.82	**0.88**	2.54

B. Overall Performance on Small Graphs

We compare HyTGraph with in-memory processing frameworks on small graphs, using both a power-law graph (Orkut) and a uniform road network (Road-USA), to analyze the performance trade-off between out-of-memory and in-memory processing, as shown in Table IV. In addition to cuGraph, UVM-based Grus and ImpTM-UM degrade to in-memory processing as most data can be cached in GPUs. Subway also switches to in-memory processing by copying the data entirely to GPU. In contrast, we disable the graph caching function in HyTGraph to evaluate its performance as a fully out-of-memory framework. Our findings reveal that Grus, Subway, ImpTM-UM, and cuGraph achieve the best results across various configurations on these small graphs. On the Road-USA graph, the disadvantages of out-of-memory processing become more evident due to its disproportionately long absolute runtime relative to the graph size. This is primarily attributed to the small average degree and large diameter of the road network, which result in a higher number of iterations. Algorithms such as CC and SSSP require 5,000–6,000 iterations to converge, each involving substantial fragmented edge data accesses (each access involves up to 9 neighborhoods per). EMOGI with zero-copy access exhibits inferior performance in this scenario, mainly due to PCIe underutilization caused by transferring sliced data. In addition, the performance on the road network is also affected by load balancing. We observe that in-memory processing frameworks with fine-grained workload balancing, such as cuGraph and Subway (which leverages Tigr's degree-optimized load balancing [37]), outperform other frameworks in different cases. Notably, Galois outperforms GPU baselines for the BFS algorithm on the Road-USA. This demonstrates that CPU-based frameworks still hold advantages in certain scenarios.

C. Overall Performance on Large Graphs

Comparison with ExpTM-F, Subway, and EMOGI. Table IV shows the overall results. Due to the heavy redundant transfer, ExpTM-F shows inferior performance. The speedup of HyTGraph over ExpTM-F ranges from 2.81X (for PageRank on FK) to 28.52X (for BFS on UK) with an average of 9.89X. Neither Subway nor EMOGI is always better than the other. The speedup of HyTGraph over Subway ranges from 2.91X (for SSSP on SK) to 10.27X (for BFS on UK) with an average of 5.02X. Subway's critical performance bottleneck lies in its heavy CPU-based compaction and preprocessing (For SSSP algorithm, the preprocessing and compaction overhead account for 46.9%-74.9% of the total runtime). On CC, SSSP, and PageRank, HyTGraph is faster than EMOGI by 2.01X on average, with its speedups ranging from 1.24X to 7.91X. With the help of zero-copy access, EMOGI achieves significant performance improvement on low-activeness subgraphs. While for the high-activeness subgraphs, especially those with

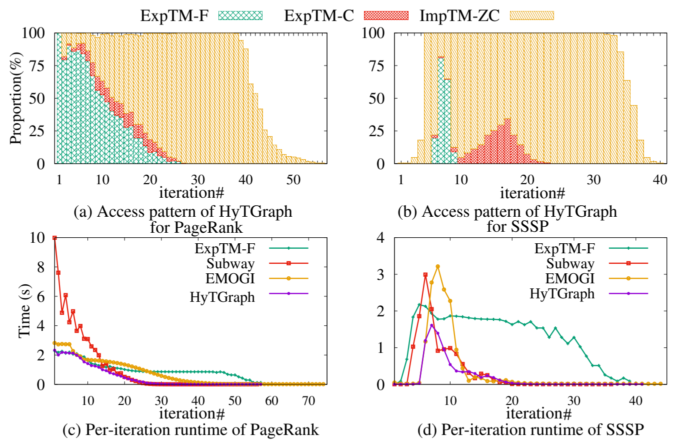

Fig. 5: Execution path of HyTGraph and per-iteration runtime comparison with ExpTM-filter, EMOGI and Subway (on FK).

TABLE V: Transfer reduction analysis.

Transfer volume / Edge volume						
Alg.	System	SK	TW	FK	UK	FS
PR	ExpTM-F	57.6X	52.4X	58.3X	30.9X	121.6X
	Subway	2.46X	**<u>5.48X</u>**	10.74X	1.79X	12.44X
	EMOGI	3.31X	20.6X	24.6X	3.81X	25.23X
	HytGraph$^-$	2.17X	10.9X	12.01X	1.68X	12.62X
	HyTGraph	**<u>1.36X</u>**	8.15X	**<u>8.82X</u>**	**<u>1.38X</u>**	**<u>10.8X</u>**
SSSP	ExpTM-F	44.3X	11.2X	28.1X	24.3X	24.1X
	Subway	4.23X	2.07X	**<u>3.32</u>**X	1.78X	3.19X
	EMOGI	3.29X	1.74X	4.81X	1.11X	2.69X
	HytGraph$^-$	3.25X	1.25X	4.60X	1.13X	2.52X
	HyTGraph	**<u>2.53X</u>**	**<u>1.10X</u>**	3.87X	**<u>1.06X</u>**	**<u>2.12X</u>**

dense and small degree vertices, EMOGI usually has low host-GPU utilization due to unsaturated memory requests. In contrast, HyTGraph achieves efficient data transfer on both high-activeness and low-activeness partitions by adopting hybrid transfer management. On BFS, HyTGraph outperforms Subway and EMOGI on SK, TW, and UK. On FK and FS, EMOGI shows better performance because most of the accesses on these two graphs are sparse. Moreover, compared with HyTGraph, EMOGI avoids the cost analysis, engine selection, and task combination. Since BFS traverses each vertex only once and thus has no cross-layer communication, we disable the vertex-centric data caching for it.

Comparison with Unified-Memory (UM)-based systems: ImpTM-UM and Grus. On the SK graph, the UM-based frameworks demonstrate superior performance for PageRank, CC, and BFS algorithms because the accessed edge-associated data can be entirely cached in the GPU memory. UM-based approaches only transfer the data once. However, when processing large graphs, the performance of ImpTM-UM degrades significantly because the implicit data transfer requires expensive page replacement and data transfer overhead. The experimental results show that on the four large graphs, HyTGraph achieves on average 3.01X and 2.55X speedups over ImpTM-UM and Grus, respectively.

Comparison with CPU-based Approach. From Table IV, we can observe that the GPU-accelerated graph processing frameworks show significant performance improvement over CPU-based Galois. Specifically, HyTGraph shows an average of average 10.34x speedup over Galois.

D. Execution Path Analysis of HyTM

To demonstrate the performance improvement of hybrid processing, we record the execution path of HyTGraph on PageRank and SSSP to show the proportion of partitions using ExpTM-filter, ExpTM-compaction, and ImpTM-zero-copy in each iteration. Figure 5 (a) shows the result on PageRank. The proportion of active partitions is high in the early iterations, HyTGraph prefers ExpTM-filter. As the algorithm converges and many vertices become inactive, the proportion of ImpTM-zero-copy increases. For SSSP in Figure 5 (b), there are few active vertices in the early and last few iterations, HyTGraph prefers ImpTM-zero-copy. When most vertices are activated in the middle iterations, HyTGraph prefers ExpTM-filter to improve the transfer efficiency. As the number of active vertex decreases, ExpTM-compaction is also used in some partitions. Figure 5 (c) and (d) show the per-iteration runtime of ExpTM-F, Subway, EMOGI, and HyTGraph. As these systems adopt different asynchronous processing strategies, the active vertex number of different systems in each iteration is not the same. HyTGraph cannot consistently outperform the others in each iteration. However, through the hybrid transfer management, HyTGraph achieves the minimum overall runtime.

E. Transfer Reduction Analysis

We analyze the effectiveness of HyTGraph's transfer reduction by comparing it with ExpTM-filter, Subway (ExpTM-compaction), and EMOGI (ImpTM-zero-copy), using PageRank and SSSP algorithms across five real-world graphs. Transfer volumes are normalized against the volume of edges. As shown in Table V, ExpTM-filter exhibits the highest transfer volume. With the help of fine-grained zero-copy access, EMOGI achieves considerable transfer reduction. However, due to the lack of effective asynchronous scheduling, the transfer volume is still large. Subway, aimed at minimizing data transfer through CPU data compaction, shows diverse performance across different algorithms due to the unstable naive asynchronous processing. Specifically, Subway excels in the PageRank algorithm, where additional computation markedly enhances convergence. However, for the value-replacement-based SSSP algorithm, Subway loses its edge due to the stale computation problem [11]. Since processing the transferred subgraph only twice each scheduling, HyTGraph's variant without data caching (HytGraph$^-$) shows marginal improvement over Subway for the PageRank algorithm. On small graphs with few partitions, e.g., the TW graph, HytGraph$^-$ even demands 2X data transfer compared to Subway. Conversely, for the SSSP algorithm, HytGraph$^-$ achieves significant transfer reduction in most scenarios through hybrid transfer management and contribution-driven priority scheduling. After incorporating vertex-centric graph caching, HyTGraph further diminishes data transmission by 7% to 37% over HytGraph$^-$. These enhancements significantly improve HyTGraph's communication efficiency.

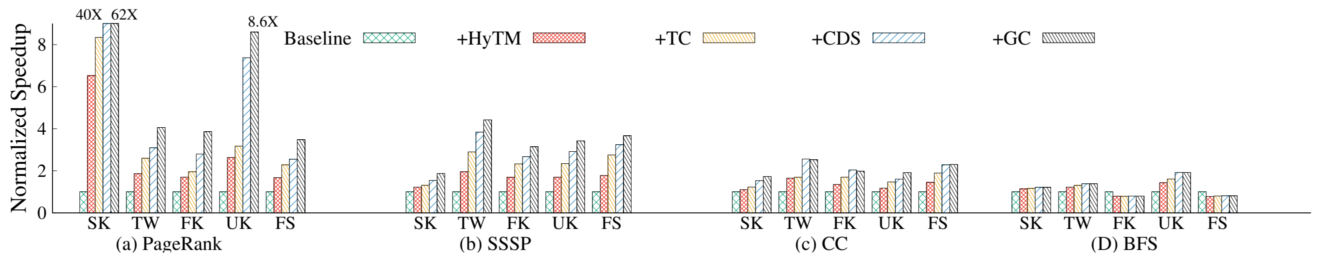

Fig. 6: Performance gain analysis of hybrid transfer management (**HyTM**), **T**ask **C**ombining (**TC**), **C**ontribution-**D**riven **S**cheduling (**CDS**), and vertex-centric **G**raph **C**aching (**GC**) .

F. Performance Improvement Analysis

To access the performance gain offered by different optimizations, our methodology initiates from a fundamental baseline utilizing ImpTM-zero-copy (EMOGI) and sequentially incorporates hybrid transfer management, vertex-centric graph caching, task combining, and contribution-driven scheduling. We selected ImpTM-zero-copy as the initial point of comparison due to its superior performance across all baselines. Figure 6 shows the normalized runtime, reflecting the performance gains attributed to each optimization.

Effectiveness of HyTM. The hybrid transfer management method brings speedups of averaging 2.87X, 1.66X, 1.34X, and 1.07X for PageRank, SSSP, CC, and BFS algorithms, respectively. For algorithms with significant variations in active vertices, such as SSSP, CC, and PageRank, HyTM demonstrates notable performance improvements. Conversely, for algorithms with a few active vertices throughout the execution, such as BFS, ImpTM-zero-copy still holds advantages. This is because it benefits from the efficient zero-copy access and avoids the overhead associated with cost analysis and task management in HyTM. On FK and FS graph, ImpTM-zero-copy even demonstrates better performance than HyTM. However, in most algorithms with diverse access patterns, HyTGraph demonstrates considerable effectiveness.

Effectiveness of task combining and contribution-driven scheduling. The task combining (TC) can bring speedups of averaging 1.28X, 1.37X, 1.19X, and 1.05X for PageRank, SSSP, CC, and BFS algorithms, respectively. The contribution-driven scheduling (CDS) further brings speedups of averaging 2.18X, 1.21X, 1.25X, and 1.06X over the hybrid processing with TC. Overall, integrating the two optimizations brings speedups of averaging 2.78X, 1.67X, 1.47X, and 1.16X over the naive hybrid transfer management method on the four algorithms. PageRank algorithm benefits most because the proposed asynchronous processing can effectively accelerate convergence by prioritizing the vertices with large contributions, i.e., the vertex value. In contrast, BFS benefits little from the two designs because each vertex is activated only once during the iterative processing, leading to a small overall transfer overhead.

Effectiveness of vertex-centric graph caching (VCGC). Since the BFS algorithm traverses each vertex only once, it can not benefit from the VCGC optimization. Therefore, we do not enable VCGC for it in the evaluation. On the other three algorithms, VCGC provides speedups ranging from 0.98X to 1.38X by reducing cross-layer communication. We

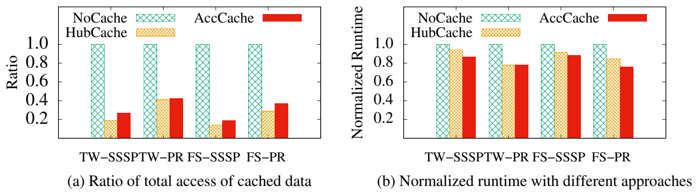

Fig. 7: Comparison with hub-vertex caching.

observe that the effectiveness of data caching differs across algorithms. For PageRank and SSSP, VCGC exhibits consistent improvements ranging from 1.13X to 1.38X. In contrast, for the CC algorithm, VCGC demonstrates slightly weaker performance on TW and FK graphs. This discrepancy arises because the runtime of these cases has short runtimes and low volumes of cross-layer communication, rendering the caching improvements insufficient to outweigh the costs associated with cache replacement. Nevertheless, as an enhancement to the hybrid transfer management method, VCGC effectively reduces transmission volumes in most scenarios. As a recommendation, VCGC is particularly advantageous for algorithms with extensive cross-layer accesses, such as PageRank and SSSP and their variants.

Comparison with hub-vertex caching. We evaluate the performance of the proposed VCGC and the hub-vertex-based caching approach by comparing: 1) the proportion of reduced edge accesses relative to the total accesses, and 2) the normalized runtime compared to HyTGraph without data caching, as shown in Figure 7. Both caching mechanisms effectively reduce the volume of edge accesses. On average, VCGC achieves a 24.2% higher reduction in edge accesses compared to hub-vertex sorting. However, in specific scenarios such as TW-PR, hub-vertex sorting performs comparably to VCGC. Specifically, VCGC and hub-vertex caching reduce edge accesses by 14.1%–40.9% (avg. 25.1%) and 17.3%–42.4% (avg. 31.0%), respectively, compared to HyTGraph without data caching. In terms of runtime improvement, the difference between the two approaches is less pronounced. VCGC achieves an average improvement of approximately 7.9% over hub-vertex caching. This is due to HyTGraph 's contribution-driven priority scheduling, which performs additional iterations for processing hub vertices, inherently reducing remote data accesses. This optimization overlaps with the caching mechanism, preventing the access reductions from fully translating into runtime gains. Looking forward, VCGC's access-frequency-based caching mechanism can be extended to a wider range of graph processing tasks, such as real-time

TABLE VI: Runtime breakdown of HyTGraph with vertex-centric graph caching on PageRank.

Engine	**Component**	time of different components (s)				
		SK	**TW**	**FK**	**UK**	**FS**
W/o caching	overall	2.85	11.5	30.2	4.71	40.9
W/ caching	Computation	1.84	8.76	21.6	4.04	29.2
	Cand. Selection	+0.07	+0.08	+0.1	+0.07	+0.09
	Cache Refreshing	+0.19	+ 0.48	+0.68	+0.44	+0.52
Benefit	-	-1.26	-3.29	-9.26	-1.17	-12.2

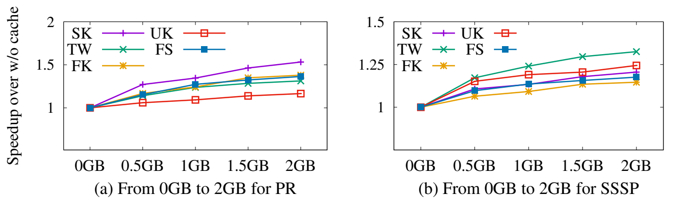

Fig. 8: Performance with varying cache sizes.

streaming graph analysis [15], graph sampling [21], and multi-hop subgraph analysis [26]. It is expected to achieve high runtime efficiency in these applications.

Overhead of graph caching. Table VI presents the overhead associated with graph caching, including the hotness-based candidate selection and Vertex-centric cache refreshing. We also compare the results with the performance gains brought by graph caching on PageRank. We observe that the time dedicated to candidate selection and cache refreshing is small, accounting for approximately 3% to 12% of total runtime. Compared to the benefit of optimizing cross-layer accesses, the total overhead of cache management is also minor, ranging from 5% to 42% of performance gain. With an appropriate cache refresh configuration, HyTGraph typically caches the most correct vertices after a single (or two for TW and FK graph) cache refreshing cycle. In summary, for graph iterative algorithms with extensive cross-iteration repeat vertex accesses (e.g., PageRank), GPU graph caching can significantly enhance performance, with the incurred management overhead being effectively outweighed by the benefits.

G. Sensitivity Analysis

Varying cache sizes. To evaluate the impact of varying cache sizes, we run SSSP and PR on the five large graphs, starting from no hot data caching (0GB) and linearly increasing the cache size to 2GB. As shown in Figure 8. We observe that the first 0.5B of data caching yields a significant performance improvement, approximately 42% to 62% compared to the improvement of caching 2GB of data. This is because, due to the power-law distribution, frequently accessed vertices represent a small proportion of the total but account for a large number of edge accesses. As the cache size increases from 0.5GB to 2GB, the per-GB benefit of caching graph data decreases. Nevertheless, the overall performance improvement remains non-negligible.

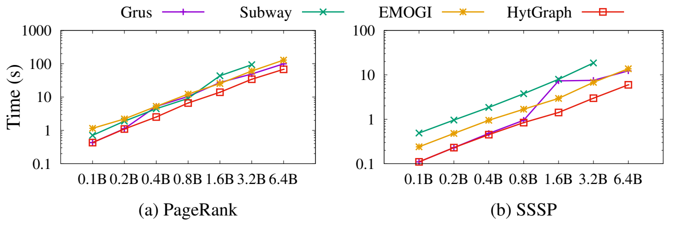

Fig. 9: Performance comparison with increasing graph size, the graphs are generated by RMAT with sizes from 0.1 Billion to 6.4 Billion (64X).

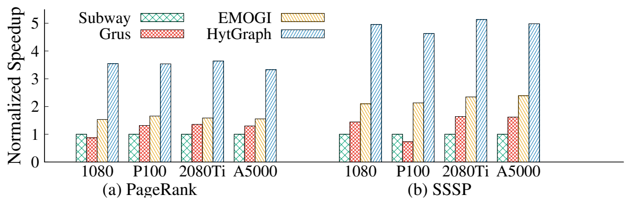

Fig. 10: Performance comparison on different GPUs (FS).

Varying graph sizes. We compare HyTGraph with Grus, Subway, and EMOGI under variable graph sizes and report the results in Figure 9. Subway encounters an integer overflow issue and fails to process graphs with 6.4 billion edges. Grus exhibits superior performance on small graphs because the data only needs to be loaded once. However, as graph sizes increase, the performance declines because the overhead of data migration of unified memory increases. We observe that most system demonstrates linear scaling with increasing graph sizes. Benefiting from the vertex-centric GPU data caching, HyTGraph achieves comparable performance with Grus and outperforms other systems on small graphs. On large graphs, Grus underperforms on the SSSP algorithm due to the increased overhead of frequent memory page swapping of low active data. In contrast, HyTGraph can efficiently process large graphs through a combination of hybrid transfer management and vertex-centric data caching. As the graph size increases from 0.1B to 6.4B (64X), the runtime of Grus, EMOGI, and HyTGraph for PageRank increases by 231.2X, 111.6X, and 157.86X, respectively. For the SSSP algorithm, the runtime of Grus, EMOGI, and HyTGraph increases by 111.8X, 57.08X, and 54.22X, respectively. Due to limited cache capacity, the performance benefits of HyTGraph's caching do not scale proportionally as the graph size increases, which makes it appear to have weaker scalability compared to EMOGI without caching. However, HyTGraph still demonstrates a significant advantage in absolute runtime. Furthermore, compared to the UVM-based page caching approach used in Grus, HyTGraph's caching method offers superior scalability.

Varying GPUs. We assess HyTGraph's performance across various platforms equipped with different GPUs, including a GTX 1080 connected to one i7-8700k CPU through PCIe3.0x8, a NVIDIA P100 connected to dual Xeon-sliver 4210 CPUs through PCIe3.0x16, a GTX 2080Ti connected to dual Xeon Silver 4210 CPUs through PCIe3.0x16, and an NVIDIA A5000 connected to dual Xeon silver 4316 CPUs through PCIe4.0x16. The FS graph serves as the input data. We normalize the runtime of all systems to Subway and present the results in Figure 10. We can observe that HyTGraph outperforms all three competitors across all platforms. For PageRank (and SSSP), HyTGraph achieves speedups of 3.4X-3.6X (4.6X-5.1X), 2.6X-4.0X (3.1X-6.3X), and 2.1-2.2X

(2.1X-2.3X) over Subway, Grus, and EMOGI, respectively.

X. Limitations and Future Work

Extending HyTGraph to multiple GPU acceleration. Currently, HyTGraph assumes the vertex data can fit into a single GPU. When processing graphs with vertex data that exceed a single GPU's memory capacity, the data can be partitioned across multiple GPUs. This approach introduces additional challenges, including managing host-GPU communication bandwidth, inter-GPU communication overhead, and achieving a balance among computation, communication, and cache utilization. We take developing multi-GPU-based HyTGraph as future work.

Adapting to GPU platforms with advanced interconnects. Recently, the hardware manufacturers have introduced advanced interconnects, such as NVIDIA NVlink [34], Intel CXL [10], and AMD Infinity Fabric [1]). These technologies facilitate the construction of larger host memory pools for scaling large graphs and enable devices and host memory to connect through faster and more heterogeneous communication links. While these advancements offer new opportunities for processing large-scale graphs, they also present challenges in optimizing irregular data transfers across complex communication links. Extending HytGraph to support these emerging interconnect technologies is part of our future work.

XI. Related Work

In-GPU-memory graph processing. The high parallelism of GPU has attracted great attention [12], [19], [20], [30], [46], [49], [51] in graph processing community. Cusha [23] uses two novel data structures, named GShards and CW, to avoid non-coalesced memory access. Gunrock [46] performs computation on the frontier with data-centric abstraction. Tigr [37] proposes a virtual transformation to transform skewed graphs into virtual vertices for load-balancing. SEP-Graph [42] optimize execution paths by adaptively switching Sync/Async, Push/Pull, and data-driven/topology-driven modes.

Out-of-core GPU graph processing. GPU-accelerated graph processing has attracted extensive attention. Besides the systems mentioned above [13], [18], [31], [38]–[40], [43], [50], recent studies also propose CPU-GPU co-processing to accelerate large graphs computation [14], [28]. However, the CPU-based low-activeness subgraph processing may become a new bottleneck. Besides graph processing, researchers have also investigated GPU-accelerated pattern matching on large graphs [7], [17] that optimize communication by sharing execution or combining zero-copy access and unified virtual memory.

XII. Conclusion

We present HyTGraph, a highly efficient GPU-accelerated graph processing framework by adaptively switching transfer management methods involving explicit transfer management and implicit transfer management. This hybrid approach maximizes the host-GPU bandwidth and is necessary to achieve the shortest overall execution time. Moreover, HyTGraph provides a vertex-centric graph caching method that further reduces communication through data transfer reusing. Our intensive experiments show the high effectiveness of HyTGraph.

References

[1] Amd infinity fabric link. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/other/56978.pdf. Accessed: 2024-10-11.

[2] The koblenz network collection. http://konect.uni-koblenz.de/. Accessed: 2021-09-01.

[3] Laboratory for web algorithmics. http://law.di.unimi.it/. Accessed: 2021-09-01.

[4] N. Agarwal, D. W. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 607–618. ACM, 2015.

[5] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pages 136–150. ACM, 2017.

[6] 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, Austin, TX, USA, February 4-8, 2017*, pages 235–248. ACM, 2017.

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

[8] Nvidia cub, 2023. https://github.com/NVIDIA/cub.

[9] Rapids cugraph, 2023. https://docs.rapids.ai/api/cugraph/stable/.

[10] CXL. Compute express link specification revision 1.1. https://www.computeexpresslink.org/, 2022.

[11] W. Fan, P. Lu, X. Luo, J. Xu, Q. Yin, W. Yu, and R. Xu. Adaptive asynchronous parallelization of graph algorithms. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD 2018, Houston, TX, USA, June 10-15, 2018*, pages 1141–1156. ACM, 2018.

[12] A. Gaihre, Z. Wu, F. Yao, and H. Liu. XBFS: exploring runtime optimizations for breadth-first search on gpus. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2019, Phoenix, AZ, USA, June 22-29, 2019*, pages 121–131. ACM, 2019.

[13] P. Gera, H. Kim, P. Sao, H. Kim, and D. A. Bader. Traversing large graphs on gpus with unified memory. *Proc. VLDB Endow.*, 13(7):1119–1133, 2020.

[14] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 345–354. ACM, 2012.

[15] S. Gong, C. Tian, Q. Yin, W. Yu, Y. Zhang, L. Geng, S. Yu, G. Yu, and J. Zhou. Automating incremental graph processing with flexible memoization. *Proc. VLDB Endow.*, 14(9):1613–1625, 2021.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30. USENIX Association, 2012.

[17] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD 2020, Portland, OR, USA, June 14-19, 2020*, pages 1067–1082. ACM, 2020.

[18] W. Han, D. Mawhirter, B. Wu, and M. Buland. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 429–442. USENIX Association, 2019.

[19] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High Performance Computing - 2007, 14th International Conference, Goa, India, December 18-21, 2007, Proceedings*, volume 4873 of *LNCS*, pages 197–208. Springer, 2007.

[20] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 267–276. ACM, 2011.

[21] H. Hu, F. Liu, Q. Pei, Y. Yuan, Z. Xu, and L. Wang. λgrapher: A resource-efficient serverless system for GNN serving through graph sharing. In T. Chua, C. Ngo, R. Kumar, H. W. Lauw, and R. K. Lee,

editors, *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, pages 2826–2835. ACM, 2024.

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

[23] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *HPDC'14*, pages 239–252, 2014.

[24] 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, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 447–461, 2016.

[25] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46. USENIX Association, 2012.

[26] C. Li, H. Chen, S. Zhang, Y. Hu, C. Chen, Z. Zhang, M. Li, X. Li, D. Han, X. Chen, et al. Bytegraph: a high-performance distributed graph database in bytedance. *Proceedings of the VLDB Endowment*, 15(12):3306–3318, 2022.

[27] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on gpus. In J. Kern and J. S. Vetter, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 68:1–68:12. ACM, 2015.

[28] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *2017 USENIX Annual Technical Conference, ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 195–207. USENIX Association, 2017.

[29] 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, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

[30] D. Merrill, M. Garland, and A. S. Grimshaw. High-performance and scalable GPU graph traversal. *ACM Trans. Parallel Comput.*, 1(2):14:1–14:30, 2015.

[31] S. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W. Hwu. EMOGI: efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 14(2):114–127, 2020.

[32] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471. ACM, 2013.

[33] NVIDIA. Nvidia a100 tensor core gpu. https://www.nvidia.com/en-us/data-center/a100/, 2022.

[34] NVIDIA. Nvidia h100 tensor core gpu. https://www.nvidia.com/en-us/data-center/h100/, 2022.

[35] NVIDIA. Nvidia tesla p100. https://www.nvidia.com/en-us/data-center/tesla-p100/, 2022.

[36] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 472–488. ACM, 2013.

[37] A. H. N. Sabet, J. Qiu, and Z. Zhao. 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, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 622–636. ACM, 2018.

[38] 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, Heraklion, Greece, April 27-30, 2020*, pages 12:1–12:16. ACM, 2020.

[39] 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, Austin, TX, USA, November 15-20, 2015*, pages 28:1–28:12. ACM, 2015.

[40] 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, Lemont, IL, USA, August 9 - 12, 2021*, pages 41:1–41:10. ACM, 2021.

[41] K. Vora. LUMOS: dependency-driven disk-based graph processing. In *USENIX ATC 2019*, pages 429–442. USENIX Association, 2019.

[42] 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 2019, Washington, DC, USA, Feb 16-20, 2019*, pages 38–52. ACM, 2019.

[43] 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.*, 18(2):22:1–22:25, 2021.

[44] Q. Wang, X. Ai, Y. Zhang, J. Chen, and G. Yu. Hytgraph: Gpu-accelerated graph processing with hybrid transfer management. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 558–571. IEEE, 2023.

[45] Q. Wang, Y. Zhang, H. Wang, L. Geng, R. Lee, X. Zhang, and G. Yu. Automating incremental and asynchronous evaluation for recursive aggregate data processing. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD 2020, Portland, OR, USA, June 14-19, 2020*, pages 2439–2454. ACM, 2020.

[46] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: a high-performance graph processing library on the GPU. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pages 239–252. ACM, 2016.

[47] 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.*, 25(8):2091–2100, 2014.

[48] Y. Zhang, V. Kiriansky, C. Mendis, S. P. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, December 11-14, 2017*, pages 293–302. IEEE Computer Society, 2017.

[49] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu. Digraph: An efficient path-based iterative directed graph processing system on multiple gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 601–614. ACM, 2019.

[50] L. Zheng, X. Li, Y. Zheng, Y. Huang, X. Liao, H. Jin, J. Xue, Z. Shao, and Q. Hua. Scaph: Scalable gpu-accelerated graph processing with value-driven differential scheduling. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 573–588. USENIX Association, 2020.

[51] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distributed Syst.*, 25(6):1543–1552, 2014.

[52] X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 375–386. USENIX Association, 2015.

Qiange Wang received the PhD degree in computer science from Northeastern University, China, in 2022. He is currently a postdoctoral research fellow at the National University of Singapore. His research interests include distributed graph processing, learning, and management systems.

Xin Ai is currently working toward the PhD degree in computer science with Northeastern University. His research interests include parallel and distributed graph computing and learning system.

Yongze Yan received the Master degree in computer science from the Northeastern University of China, Shenyang, in 2023. He is currently working toward the PhD degree. His major research interests include GPU data processing and database on emerging hardware.

Shufeng Gong received the PhD degree in computer science from Northeastern University, China, in 2021. He is currently a lecturer with Northeastern University, China. His research interests include cloud computing, distributed graph processing, and data mining.

Yanfeng Zhang received the PhD degree in computer science from Northeastern University, China, in 2012. He is currently a professor with Northeastern University, China. His research consists of distributed systems and big data processing. He has published many papers in the above areas. His paper in SoCC 2011 was honored with ”Paper of Distinction”.

Jing Chen received the Master degree in computer science from Northeastern University, China, in 2022. Her research interests include GPU graph processing systems.

Ge Yu received the PhD degree in computer science from the Kyushu University of Japan, in 1996. He is now a professor with Northeastern University, China. His current research interests include distributed and parallel systems, cloud computing, big data management, and blockchain techniques and systems. He has published more than 200 papers in refereed journals and conferences. He is the CCF fellow, the IEEE senior member, and the ACM senior member.