HyTGraph: GPU-Accelerated Graph Processing with Hybrid Transfer Management

Qiange Wang*, Xin Ai*, Yanfeng Zhang✉ , Jing Chen, Ge Yu

School of Computer Science and Engineering
Northeastern University, Shenyang, China
{wangqiange,aixin0,chenjing}@stumail.neu.edu.cn; {zhangyf,yuge}@mail.neu.edu.cn

Abstract—Processing large graphs with memory-limited GPU needs to resolve issues of host-GPU data transfer, which is a key performance bottleneck. Existing GPU-accelerated graph processing frameworks reduce the data transfers 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 access with zero-copy or unified-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.

In this work, we propose a hybrid transfer management approach to take the merits of both the two approaches at runtime, with an objective to achieve the shortest execution time in each iteration. Based on the hybrid approach, we present HyTGraph, a GPU-accelerated graph processing framework, which is empowered by a set of effective task scheduling optimizations to improve the performance. Our experimental results on real-world and synthesized graphs demonstrate that HyTGraph achieves up to 10.27X speedup over existing GPU-accelerated graph processing systems including Grus, Subway, and EMOGI.

Index Terms—GPU, Graph processing, Hybrid transfer management

I. INTRODUCTION

Analyzing large-scale graph data plays an important role in real-world applications, including geo-information mining, social network analysis, and business association analysis.

Compared with the shared-memory-based frameworks and the shared-nothing-based frameworks, GPU-based graph processing attracts more attention for its high memory bandwidth and massive parallelism [17], [32], [37], [40], [45]. Unfortunately, GPU's limited device memory can only accommodate a small set of real-world graphs. When the size of the input graph exceeds the GPU memory capacity (*memory oversubscription*), existing GPU-based systems fail to work (e.g., Medusa [45], CuSha [17], Gunrock [40], Tigr [32], SEP-Graph [37], etc).

Recently, researches [11], [12], [22], [25], [33]–[35], [38], [44] focus on supporting GPU-accelerated graph processing to take advantage of both the high-performance GPU graph processing and the large host memory for storing the large-scale graphs. Similar to that out-of-core graph processing

TABLE I: Advances from NVIDIA P100 to H100.

GPUs	Mem. bdw.	PCIe x16 bdw.	Mem. bdw/ PCIe. bdw
P100 [29] (2016)	732GB/s	16GB/s (Gen3)	**45.8X**
V100 [30] (2017)	900GB/s	16GB/s (Gen3)	**50X**
A100 [27] (2020)	1.9TB/s	32GB/s (Gen4)	**48.6X**
H100 [28] (2022)	3TB/s	64GB/s (Gen5)	**48X**

[19], [31], [36], [47], the major challenge for GPU-accelerated graph processing is the low computation resource utilization caused by the extensive data movement overhead between GPU and host memory. Compared to the high-speed global memory access bandwidth in GPU, the host memory and GPU are connected with a slow PCIe interface, which can be an order of magnitude slower. For example, the host-GPU bandwidth via PCIe 3.0 can be limited to be 16GB/s (12.3GB/s in practice) [25]. Moreover, the development of new generation PCIe has not narrowed the bandwidth gap. Even though the PCIe bandwidth has been improved in past years (from 16GB/s to 64GB/s), the bandwidth gap between GPU memory and PCIe is still very large as shown in Table I.

To reduce the data movements between GPU and host memory, the existing GPU-accelerated frameworks [11], [14], [25], [33], [34], [38], [44] track the evolving active vertices during the iterative processing. Considering a vertex-centric graph processing, where the computation is performed in a series of iterations, in each iteration, the algorithm takes only the vertices updated by the previous iteration as input (i.e., active vertices), updates their out-going neighbors and marks the neighbors whose values have been updated as the active vertices in the next iteration. During the iterative processing, only the out-going edges of the active vertex (i.e., active edges) need to be accessed. Following the existing frameworks [11], [14], [25], [33], [34], [38], [44], we assume that the vertex-associated data (including vertex value, neighbor index, and activity status) can be resident in the GPU memory and the edge-associated data (including edges and edge weights) can entirely fit into the host memory. During the iterative processing, the active subgraph containing active edges must be transferred to the GPU memory.

According to the way of reducing host-GPU data transfers, the existing frameworks can be classified into two categories: **Explicit** (active subgraph) **Transfer Management (ExpTM)** based frameworks [14], [33]–[35], [44] and **Implicit** (active

TABLE II: 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)

subgraph) **Transfer Management (ImpTM)** based frameworks [11], [25], [38].

With the ExpTM approach, the programmers have to manually manage the active subgraph transfer. In ExpTM-based frameworks, the oversized graph is partitioned into smaller subgraphs that can fit into GPU device memory. Before being transferred to GPU through explicit memory copy engine (cudaMemcpy), the subgraphs have to pass through a CPU-based redundancy removal module to remove inactive edges. According to the working mode, this approach can be either filter-based [14], [18], [34] or compaction-based [33], [35], [44], and the transfer reduction performance is determined by the power of removal module.

Recently, a more general solution, ImpTM-based approaches have become available [11], [25], [38]. Rather than explicitly managing the data movements of active subgraphs. ImpTM-based frameworks allow GPU programs to access the active edges in the host memory in an on-demand mechanism [4], [5], [11], [25]. Using ImpTM requires little engineering work, we can directly extend the single GPU frameworks into an out-of-core one by managing the host-resident edge data with unified-memory [11], [38] or zero-copy memory [25]. Then during the iterative processing, the memory unit containing active edges can be implicitly transferred to the GPU memory without programmers' manual management. Since ImpTM approaches rely on the system-provided memory access mechanism, its transfer efficiency is sensitive to the graph access pattern. Recent research [25] shows that the performance gap can reach more than three times.

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 a GPU-accelerated graph processing framework based on a single approach, the performance is often suboptimal. We show the performance comparison of Subway [33] (a ExpTM-compaction-based framework) and EMOGI [25] (an ImpTM-zero-copy-based framework). Table II shows that on sk-2005 graph [2], EMOGI outperforms the Subway on Single Source Shortest Path algorithm (SSSP) , but it losses on PageRank. In contrast, for PageRank algorithm, Subway beats EMOGI on SK dataset [2], but losses on UK dataset [2].

In this paper, we present a **Hybrid Transfer Management** approach (**HyTM**). Unlike prior frameworks that use either **ExpTM** or **ImpTM**, our hybrid approach combines ExpTM and ImpTM to maximize the performance. In the preprocessing stage, HyTM partitions the graph as ExpTM does. Then during the iterative processing, it estimates ExpTM cost and ImpTM cost on-the-fly by analyzing the edge ac-
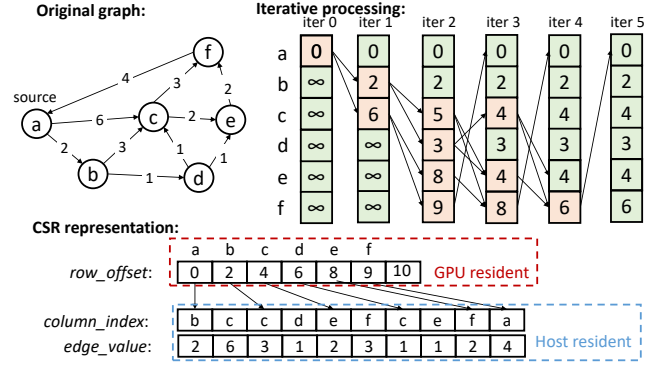

Fig. 1: An example of vertex-centric SSSP computation starting from source a. The orange box represents the active vertex and the green box represents the inactive vertex. The input graph is organized into CSR, whose vertex-associated data is resident in the GPU and the edge-associated data is resident in the host memory.

cess pattern of each partition and chooses the most cost-efficient transfer approach. Based on the **HyTM**, we propose HyTGraph, a GPU-accelerated graph processing system with flexible asynchronous task scheduling. Unlike prior frameworks [14], [33], [34], [44] that simply process the loaded subgraphs multiple times, HyTGraph adopts a contribution-driven priority scheduling method. Through a lightweight graph reorganization, HyTGraph can gather and prioritize the vertices that have a large distribution to convergence.

We have made the following contributions in this paper.

- Providing insights into the two existing approaches. We conduct a comprehensive study on the performance merits and limits of the two transfer management approaches (ExpTM and ImpTM).
- Proposing a hybrid transfer management framework. We introduce a hybrid transfer management method to maximize the performance by taking the merit of both ExpTM and ImpTM.
- Delivering a GPU-accelerated graph processing system. Based on the hybrid transfer management method, we design and implement HyTGraph, a transfer-efficient GPU-accelerated graph processing system with flexible asynchronous task scheduling to enable high performance.

We evaluate HyTGraph on both real-world and synthesized graphs. The experimental results show that HyTGraph outperforms the state-of-the-art systems, i.e., on average 4.61X speedup over Subway [33], 2.37X speedup over Grus [38] and 1.74X speedup over EMOGI [25].

II. BACKGROUND

A. Vertex-Centric Graph Processing and Active vertices

Vertex-centric programming [13], [23] has been widely adopted in Graph processing frameworks, for its simplicity, high scalability, and powerful expression ability. It defines a generic function that defines the behavior of a vertex and its neighbors. Considering the message passing direction, the function can be either pull-based or push-based [37]. During

the computation, this function is evaluated on all input vertices iteratively until the algorithm reaches convergence. Figure 1 illustrates a push-based example of SSSP, an algorithm to find the shortest paths from a given source vertex to all the other vertices. It starts from the source vertex a, where the initial distance is set to 0. In each iteration, each input vertex sends its current shortest distance to the outgoing neighbors, and the neighbor receiving messages will update its shortest distance as the shortest one. When no vertices is updated, the algorithm converges. We can observe that, during the iterative computation, only the vertices updated by the previous iteration (active vertices) need to be processed and its number increases with the message scatter from the source vertex and decreases as most vertices converge.

The graph processing which processes graph data iteratively has a special memory access pattern. The edge data that requires substantial memory footprint is read-only, and the vertex data that requires small memory footprint is read-write. In a GPU-accelerated platform where the input graph exceeds the GPU memory capacity, placing the relatively small vertex data in GPU and accessing the required edge data on demand from host memory is a worth trying approach. Firstly, The edge data transfer is easier to manage than the vertex data transfer because the edge data is read-only, requiring only one-way communications (host-to-GPU). Secondly, in real-world graphs, the number of vertex is often orders-of-magnitude less than the number of edge. Even a commonly used 16GB GPU can still process a large graph with hundred-millions of vertices and tens of billions of edges. As this design needs to retransfer the edge-associated data multiple times, adopting additional transfer management module to reduce the inactive edge transfers is critical to performance.

B. *ExpTM* Approaches

ExpTM-filter. GraphReduce [34], GTS [18], and Graphie [14] adopt filter-based method to reduce the inactive subgraph transfer. They monitor the active edges of the partitioned subgraphs in each iteration and transfers only those partitions that contain active edges. Figure 2 (a) provides an illustrative example. This method only filters out partitions that do not contain active edges without doing additional processing, so the entire partition will be directly transferred to GPU even if only one edge is active. When the proportion of active edges in a partition is low, the volume of redundant data transfer will be large.

ExpTM-compaction. In contrast, some frameworks [33], [35], [44] introduce CPU-assisted compaction to reduce redundant data transfers. Before transferring a partitioned subgraph to GPU, these frameworks use CPUs to remove the inactive edges and compact the remaining edges into a continuous memory space to leverage explicit memory copy. Figure 2 (b) shows an illustrative example. Subway [33] is a typical ExpTM-compaction-based system. In each iteration, it compacts all the active edges into a new graph and transfer it to GPU for parallel processing. Compared with the filter-based frameworks [14], [34], compaction-based frameworks

can minimize the data transfers by removing all inactive edges. But at the cost, it involves heavy CPU and main memory read/write overhead.

C. *ImpTM* Approaches

ImpTM-unified-memory. The unified-memory defines a managed memory space in which both GPU and CPU can observe a single address space with a coherent memory image [11], [38]. The memory pages (4KB in default) containing the requested data are automatically migrated to the devices, and the subsequent accesses to the same page will read data from the GPU memory without additional data transfers. When the memory footprint of the kernel is larger than the GPU memory, some pages may need to be evicted from the GPU to make room for the new pages. Figure 2 (c) shows an illustrative example. Notice that the "automated migration" cost is not free. When a requested memory page is not in the GPU memory, a page fault is triggered, which requires not only data transfer but also heavy Translation Lookaside Buffer (TLB) invalidation and page table updating overhead [25].

ImpTM-zero-copy. In contrast, zero-copy memory access is a more lightweight approach. Zero-copy maps pinned host memory to GPU address spaces, allowing GPU programs to directly access the host memory through the Transaction Layer Packet (TLP) of PCIe [25]. Compared with unified-memory, zero-copy provides much fine-grained access granularity. By the PCIe 3.0 specification, each TLP can process at most 256 outstanding memory requests simultaneously, and each request can carry 32/64/96/128-byte [25] data according to the size of accessed data. Therefore, zero-copy allows the programs to access the edges of multiple randomly distributed active vertices simultaneously, and each vertex occupies one or several memory requests. In addition, zero-copy requires less transferring overhead than unified-memory based frameworks because it requires no additional page migration overhead. As a sacrifice, the zero-copy method cannot provide the data reuse function. Multiple accesses to the same data will cause multiple separate data transfers.

III. ANALYSIS OF EXISTING APPROACHES: A MOTIVATING STUDY

In this section, we experimentally analyze the existing approaches with two graph algorithms SSSP and PageRank, because they have two typical active vertices change patterns (increase then decrease, and monotone decrease). The details of the used graphs, test platform, and system configurations are given in Section VII-A.

A. Analysis of ExpTM

ExpTM-filter. As mentioned above, the filter-based ExpTM has a large volume of redundant transfers even if the proportion of active edge is low. We run PageRank and SSSP on friendster-konect [1] graph to explore the redundant data transfer problem, the partition number is set to 256. Figure 3 (a) shows the proportion curves of active edges and partitions
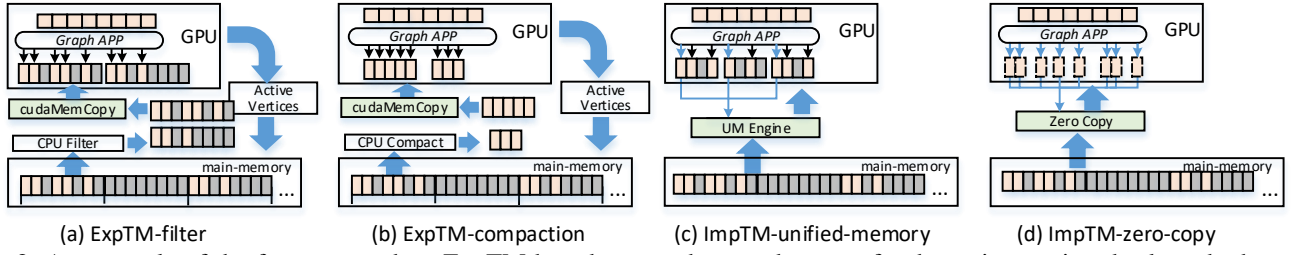
Fig. 2: An example of the four approaches, ExpTM-based approaches need to transfer the active vertices back to the host side for removing the inactive edges, ImpTM-based methods does not need this process. The thin blue arrow, thin black arrow, and thick blue arrow represent the remote memory access, local memory access, and host-GPU data transfer, respectively.
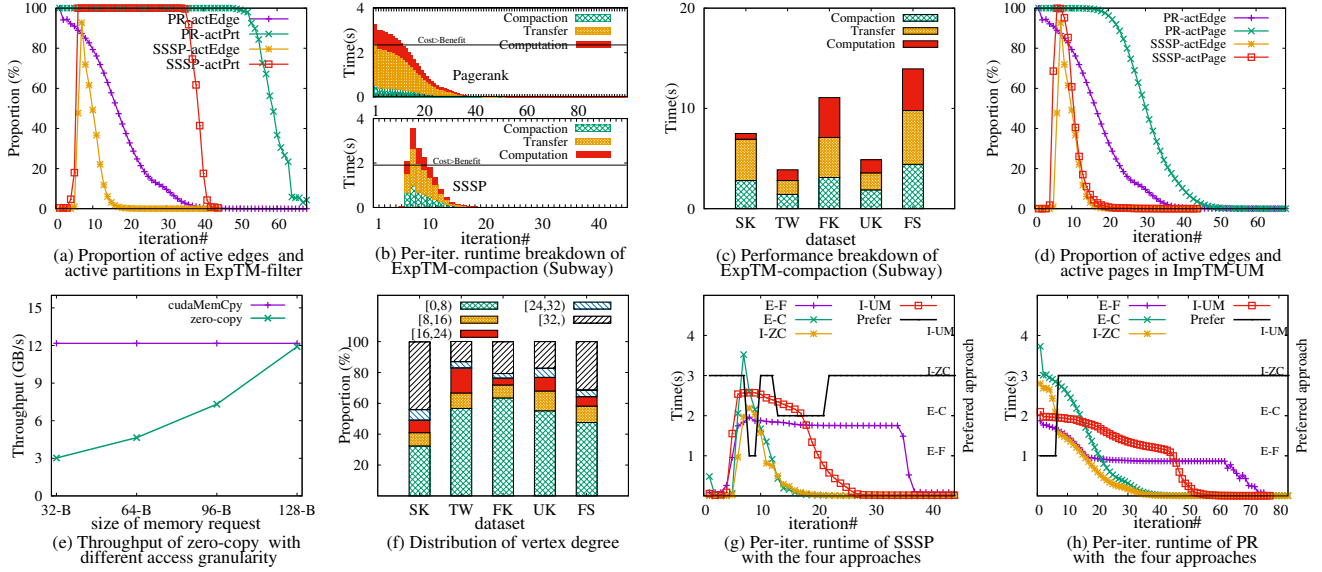

Fig. 3: Performance analysis of four engine of the two approaches.

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. Therefore, ExpTM-filter is inefficient when there are few active edges in the partition. While, when the proportion of active edge is large, ExpTM-filter method shows advantages, because it can fully utilize the PCIe bandwidth with cudaMemcpy.

ExpTM-compaction. The compaction-based ExpTM achieves significant transfer reduction and can leverage the efficient explicit memory copy. But it involves heavy additional active edge compaction overhead, which is positively correlated to the proportion of active edges. As pointed out by Subway [33], when the proportion of active edges is large (e.g., 80%), the cost of compaction can even outweigh the benefit of transfer reduction [33], Figure 3 (b) illustrates the per-iteration runtime breakdown of Subway (a ExpTM-compaction based framework) and indicates when the costs outweigh the benefits. Figure 3 (c) illustrates the overall performance breakdown of SSSP algorithm on Subway, we remove its preprocessing stage and show only the execution time. We can observe that on all five datasets, the compaction stage accounts for 34.5% of the overall runtime.

B. Analysis of ImpTM

ImpTM-unified-memory. Unified-memory is not an efficient way of handling graph algorithms. First, the cost of "automated migration" is high, due to heavy TLB invalidation overhead and page table updating overhead [25], the peak bandwidth of unified-memory can only reach 73.9% of that of explicit memory copy (cudaMemcpy) [25]. Second, the graph algorithms usually have poor temporal locality [25], [35]. When accessing the vertex with only several or dozens of neighbors, the 4KB memory page may still contain non-negligible inactive data [11], [25]. Figure 3 (d) shows the proportion of the active edges and the 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. For these two reasons, the Unified-memory-based ImpTM-zero-copy shows poor transfer efficiency on large graphs, no matter the proportion of active edge is high or low. However, the UM-based method will have good performance when the graph size is small enough to fit into GPU memory because the graph can be fully cached in the GPU memory after being transferred once.

ImpTM-zero-copy. The key challenge of implementing efficient zero-copy based graph processing is fully utilizing the

PCIE bandwidth. As pointed out by EMOGI [25], saturating most of the 256 memory requests in each TLP with 128-byte data is necessary for maximizing the PCIe bandwidth utilization. In addition to the payloads of memory requests, the TLP also includes a header field to maintain the necessary control information. A smaller memory request size means that PCIe needs to use more TLPs to process the same amount of data, and thus wastes more bandwidth on transferring the header fields. Figure 3 (e) shows the throughput of zero-copy under different memory request granularity (from 32 byte to 128 byte). We can observe that, when the memory request size is 128-byte, the zero-copy access can achieve almost the same performance as `cudaMemcpy` (the maximum PCIe utilization). While, when the access granularity is set to 32-byte, the throughput decreases significantly. To achieve the maximum bandwidth utilization, EMOGI [25] proposes merged and aligned access with which each warp of threads access consecutive neighbors of one vertex in a 128-byte cache line size from the edge-associated array. In this way, the neighbors of high-degree vertices can be accessed with consecutive and saturated memory requests. However, guaranteeing most of the memory requests reach 128-byte is challenging. Assuming each vertex occupies 4-byte, we need 32 neighbors per vertex to saturate the 128-byte memory requests. In real-world graphs, the number of neighbor is often less than this value due to the power-law property. Figure 3 (f) illustrates the distribution of vertex degrees of five real-world graphs used in this paper. Most vertices (on average 74.7%) have less than 32 neighbors, and 51.1% of them have less than 8 neighbors. Zero-copy based method has unstable performances on real world graphs, it prefers subgraphs with few active vertices and large average degrees.

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 [1] with two typical graph algorithms (the traversal algorithm SSSP and the iterative algorithm PageRank [37]) in Figure 3 (f) and (g). We implement ExpTM-filter (E-F), ImpTM-unified-memory (I-UM), and ImpTM-zero-copy (I-ZC) with SEP-Graph's processing kernel [37]. Since the GPU does not write data back to host memory, we open the `cudaMemAdviseSetReadMostly` optimization for ImpTM-unified-memory (the evicted memory pages will be discarded instead of written back to host memory). We use Subway [33] as the ExpTM-compaction (E-C), because of its highly-optimized CPU compaction engine and kernel function from Tigr [32]. All the approaches are configured with synchronous processing to ensure that the number of active vertices in each iteration is roughly the same.

We use a "**Prefer**" curve to indicate the winner in each iteration. By referring to the proportion curves of active edges of SSSP and PageRank in Figure 3 (a), we can observe that when the proportion of active edges is large, ExpTM-filter has better performance because it has high bandwidth utilization (with `cudaMemcpy`) and requires no additional
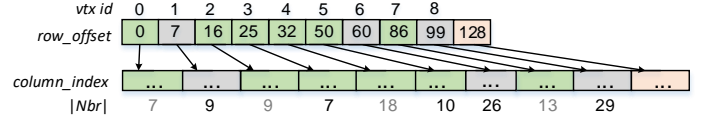

Fig. 4: A toy graph with 9 vertices and 128 edges in CSR. The graph is divided into two subset, each of which containing 64 edges. The numbers below are the number of neighbors.

CPU processing overhead. When the proportion of active edge is small, ImpTM-zero-copy shows better performance than the others in most iterations because it can transfer the neighbors needed by the active vertices with fine-grained memory requests. For SSSP algorithm, ExpTM-compaction shows better performance than ImpTM-zero-copy on some iterations. This can be attributed to the unstable performance of zero-copy under different vertex degrees. As mentioned above, the performance of zero-copy is not only related to the proportion of active edges, but also related to the number of active vertices. When the number of active edges is fixed, a large number of active vertices means that zero-copy has to use more unsaturated memory requests to process the data and thus results in more TLPs. Figure 4 shows a toy graph with 9 vertices and 128 edges. We divide the graph into two subsets (in green and gray), each of which has 64 neighbors. The two subgraphs have the same proportion of active edges (0.5) when being activated. When the subgraph with 6 vertices (in green) is activated, zero-copy has to use 6 memory requests. When the subgraph with 3 vertices is activated, zero-copy only needs 3 memory requests. This causes zero-copy performance to be unstable, even if their proportions of active edge are the same. Therefore, neither ExpTM-compaction nor ImpTM-zero-copy shows consistently better performance than each other.

In summary, although existing approaches significantly reduce the data transfers, the performance is still suboptimal. Most of them can only adapt to one or several cases.

D. Summary of Existing Systems

In Table III, We summarize these approaches and their representative systems. We also list their strengths, weaknesses, and preferred subgraph. In addition to the systems [11], [25], [33], [34] mentioned above, Scaph [44] and Ascetic [35] adopt ExpTM-compaction. Different from Subway, Scaph [44] performs compaction on the partitioned graph. It distinguishes the partitions with a small proportion of active edges, and compacts them for the subsequent GPU processing. In contrast, the partitions with a large proportion of active edges will be entirely loaded to GPU. Ascetic [35] divides GPU memory into a static region and an on-demand region, exploits the temporal locality across iterations for the static region, and compacts the other active subgraphs with CPU for the on-demand region. Grus [38] is an ImpTM-based framework. It 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 [12], [22] also use CPU-

TABLE III: Summary of existing systems

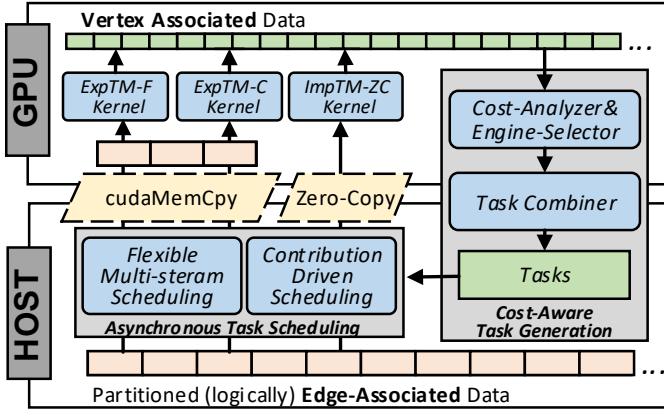| Approach	Systems	Strengths	Weaknesses	Prefer
ExpTM-filter	GraphReduce [34] Graphie [14] GTS [18]	●Less CPU overhead ●High transfer efficiency	● Redundant data transfers	●Subgraph with a large proportion of active edges
ExpTM-compaction	Subway [33] Scaph [44] Ascetic [35]	●Significant transfer reduction	● High compaction overhead	●Subgraph with a small proportion of active edges and small average degree
ImpTM-unified-memory	HALO [11] Grus [38]	● Easy to use	● Redundant data transfers ● High transfer overhead	● Small graph that can fit into GPU memory
ImpTM-zero-copy	EMOGI [25]	● Easy to use ● Fine grained memory access	● Unstable bandwidth utilization	● Subgraph with a small proportion of active edges and high average degree

Fig. 5: Overview of HyTGraph.

GPU collaboration to accelerate graph processing. We will review these works in Section IX.

IV. HYTGRAPH OVERVIEW

We present HyTGraph, a GPU-accelerated graph processing framework that adopts hybrid transfer management (HyTM) to maximize performance. HyTGraph organizes the graph into CSR structure, whose neighbor index array is resident in the GPU global memory, and edge-associated arrays (neighbor array and edge-weight array) are logically partitioned on the host side. Following the existing frameworks [14], [44], HyTGraph partitions the edge-associated data into N edge-balanced partitions $\{P_0, P_1 \ldots, P_{N-1}\}$ with chunk-based partitioning [44], [46], in which each P_i is a set of consecutively numbered vertices of partition i. During the iterative computation, the partitions containing active edges are scheduled with their most cost-efficient engine for the GPU computation. HyTGraph provides two functions to achieve efficient HyTM.

Cost-aware task generation. In the cost-aware task generation module, HyTGraph computes the data transfer costs of different approaches and selects the most cost-efficient engine for each partition. Based on the analysis in section III, we choose ExpTM-filter, ExpTM-compaction, and ImpTM-zero-copy as our baseline. In addition, HyTGraph provides a task combiner to merge the subgraphs (to be scheduled) into larger tasks to achieve lower scheduling overhead in the task scheduling stage.

Asynchronous task scheduling. HyTGraph introduces asynchrony to improve task scheduling efficiency. Rather than simply recompute the loaded subgraph multiple times [33], [44], HyTGraph adopts a contribution-driven priority scheduling to prioritize those partitions that contribute more to convergence. This method is based on the following observation: those vertices with large degrees often become the hub in the computation paths. To improve resource utilization, HyTGraph uses multiple CUDA streams to overlap the computation kernel, data transfer, and CPU-based active subgraph compaction.

Figure 5 shows an overview of HyTGraph. The cost-aware task generation and asynchronous task scheduling are iteratively alternating until the algorithm reaches convergence.

V. COST-AWARE TASK GENERATION

A. Cost Analysis and Engine Selection

Most of the Existing activeness-tracking-based frameworks use the activeness ratio as the metric [14], [22], [34], [44]. They evaluate the proportion of active edges on each partitioned subgraph to determine the appropriate processing engine. Such an approach provides an intuitive and lightweight distinguishing method, but is hard to adapt to HyTM approach. As discussed in Section III-C, the proportion of active edge cannot reflect the time cost of different approaches. In this work, we present a cost-aware processing engine selection method. During the iterative processing, we measure the overhead for each partition as follows.

Cost of ExpTM-filter. The ExpTM-filter based approach entirely transfers the partitions with active edge entirely to GPU device memory with explicit memory copy engine (cudaMemcpy). So it has only data transfer cost, which can be approximated by the saturated TLPs (as discussed in Section III, Figure 3 (e)). Given a partition i, the number of memory transaction 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 number of edge of partition i, d_1 represents the memory occupation of one vertex, and m represents the maximum capacity of an outstanding memory request (128-byte). Denote MR as the maximum number of an outstanding memory request in TLP ($MR = 256$ in PCIe 3.0 specification) and $\lceil \cdot \rceil$ as the round-up operation, we formalize the transfer overhead of each partition i as follow:

$$Tef_i = \left\lceil \left(\sum_{v \in P_i} D_o(v) \right) * d_1/m/MR \right\rceil * RTT, \quad (1)$$

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

Cost of ExpTM-compaction. ExpTM-compaction involves additional CPU-based compaction, so its cost consists of two parts, the data transfer overhead, and the 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 index. The CPU-based compaction is related to transfer volume and the throughput of CPU-based compaction, which can be computed with $\sum_{v \in A_i} D_o(v) * d_1 + |A_i| * d_2/Thpt_{cpt}$, where $Thpt_{cpt}$ is the throughput of CPU-based compaction. Then the cost of ExpTM-compaction can be formalized as follow:

$$Tec_i = \left\lceil \left(\sum_{v \in A_i} D_o(v) * d_1 + |A_i| * d_2 \right)/m/MR \right\rceil * RTT$$
$$+ \sum_{v \in A_i} D_o(v) * d_1 + |A_i| * d_2/Thpt_{cpt} \quad (2)$$

Cost of ImpTM-zero-copy. The ImpTM-zero-copy approach provides vertex-oriented on-demand access in a cacheline size, so each active vertex v takes one or several independent memory requests. The memory request number of vertex v can be formalized as $\lceil D_o(v) * d_1/m \rceil$. $D_o(v)$ represents the number of out-going neighbors of active vertex v. Considering that we can hardly guarantee the neighbors of all vertices start from the aligned memory position, some vertices may have the misaligned neighbor array and thus require one additional memory transaction [25]. We introduce a function $am()$, which returns 1 for the vertices requiring one additional transaction and 0 for the others[1]. Then the transfer overhead of partition i can be formalized as follow:

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

where $\left(\sum_{v \in P_i(V) \cap A_i} \lceil D_o(v) * d_1/m \rceil + am(v) \right)$ is the required memory transactions of active vertices. 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

---

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

compute RTT_{zc} for each partition as follows: $RTT_{zc} = \gamma * RTT + (1 - \gamma) * \left(\sum_{v \in A_i} D_o(v)/ \sum_{v \in P_i} D_o(v) \right) * RTT$, where $\left(\sum_{v \in A_i} D_o(v)/ \sum_{v \in P_i} D_o(v) \right)$ is the proportion of active edge. $\gamma * RTT$ represents the fixed time to process a TLP, and $(1 - \gamma) * \left(\sum_{v \in A_i} D_o(v)/ \sum_{v \in P_i} D_o(v) \right) * RTT$ represents the time related to the size of payload. By referring to [25], we set γ to 0.625.

Transfer engine selection. We need to compare Tef_i, Tec_i, and Tiz_i to choose the most cost-efficient transfer engine. While theoretically modeling the throughput of compaction operation $Thpt_{cpt}$ in Tec_i (formula 2) is challenging because ExpTM-compaction introduces parallel and random writes on the main 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 [33], where α is set to 80%. The second condition is based on the following observation from Section III: When a partitioned subgraph has few active edges but many active vertices, the average degree of these active vertices is small, and zero-copy requires multiple unsaturated memory requests to transfer the data. Therefore, compacting and transferring them with ExpTM-compaction is a better choice. In our implementation, β is set to 40%. If these conditions are not met, we compare Tiz_i and Tef_i. If Tiz_i is less than Tef_i, we choose ImpTM-zero-copy. Otherwise we choose ExpTM-filter. In the computation, the value of RTT can be arbitrarily specified, because it will be omitted during comparison.

Since the cost computation between partitions is independent, HyTGraph computes Tef_i, Tec_i, and Tiz_i and chooses the most cost-efficient transfer engine on GPU, transferring only the selection result back for the subsequent processing. This design can help reduce the burden of CPUs. We show the overall execution flow of the cost-aware engine selection in algorithm 1, line (2-13).

B. Task Combination

Another key to implementing hybrid transfer management is to determine appropriate task scheduling granularity. The existing frameworks [14], [22], [34], [44] directly use the partitioned subgraphs as scheduling unit. This method is simple but may lead to low efficiency in the task scheduling stage. If the partition size is too large, the coarse-grained cost computation may lead to inappropriate engine selection and thus affect the overall performance. In contrast, if using a small partition size, the transfer engine can be finely selected, but a large number of partitioned subgraphs may cause non-negligible scheduling overhead (e.g., kernel launches and fragmented data transfers) in the execution stage. Especially on those partitions with few active vertices, even a partition with one active vertex still needs to launch one CUDA kernel.

To achieve fine-grained engine selection and low overhead task scheduling simultaneously, HyTGraph decouples the graph partitioning and task partitioning to optimize them

Algorithm 1 Cost-aware task generation

Input: active vertex set $\{A_0, \cdots, A_{N-1}\}$ of N partitions,
Output: tasks prefer ExpTM-filter $\{Vf_0 \ldots Vf_{M-1}\}$ $(M < N)$, task prefer ExpTM-compaction Vc, and task prefer ImpTM-zero-copy Vz.

1: initialize a selection array $\{p_0, \ldots p_{N-1}\}$ 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, $\{p_0, \ldots p_{N-1}\}$ 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**

---

separately. HyTGraph partitions the graph into small partitions (32MB each partition) to provide fine-grained cost analysis. While in the iterative processing, HyTGraph packages the partitions choosing the same engine into large task units to reduce the scheduling overhead. Specifically, for partitions using ExpTM-filter, HyTGraph merges k consecutive partitions into a large one (k=4 in HyTGraph) to reduce the processing overhead (Line 15-24 in algorithm 1). For partitions using ExpTM-compaction, HyTGraph merges all their active vertices and writes their neighbor to one consecutive memory space to leverage efficient explicit memory copy (line 6 in algorithm 1). For partitions using ImpTM-zero-copy, HyTGraph merges all their active vertices (line 11 in algorithm 1) and processes them with one CUDA kernel to leverage the implicit transfer-computation overlapping of zero-copy.

VI. Asynchronous Task Scheduling

HyTGraph improves the asynchronous task scheduling from two directions: First, it accelerates convergence and reduces transfer volume through contribution-driven priority scheduling. Second, it improves resource utilization through multi-stream scheduling.

A. Contribution-Driven Priority Scheduling

Asynchronous computation allows the newly updated results to be used immediately in subsequent computation, which has been proved to be effective in GPU-based graph processing [6], [37]. Many GPU-accelerated graph processing frameworks [14], [33], [44] also adopt asynchronous processing to reduce the host-GPU data transfers. In these frameworks, the subgraphs loaded to GPU memory will be processed multiple times to squeeze all possible updates in each data transfer. However, simply processing the transferred subgraph multi-times may lead to inefficiency because these local updates may be abolished by the subsequent results from other partitions, leading to more additional computations and data transfers. This problems is known as stale computation problem [9], [39]. In the real-world experiment, we observe that the multi-round computation can even increase the transfer volume (See Section VII-D for details) in some cases. To effectively leverage the flexibility of asynchronous processing, HyTGraph adopts contribution-driven priority scheduling.

Hub-vertex-driven priority scheduling. Due to the power-law property of real-world graphs, some important vertices with high incoming/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, because of the large incoming degree, these vertices have a high probability of being activated in the iterative computation. If these vertices do not accumulate sufficient effective 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, because the hub vertices may distribute randomly among the whole graph, which makes host-GPU hub-vertex scheduling hard to design. To solve this problem, HyTGraph adopts the hub sorting method [42] to gather and sort the top 8% important vertices at the beginning of the CSR structure, where the importance score of each vertex v is measured by the following formula:

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

$D_i(v)$, $D_o(v)$, D_{imax}, and D_{omax} represent the incoming-, outgoing-, maximum incoming-, and maximum outgoing-degree, respectively. In this way, the hub vertices are gathered together, and the non-hub-vertices remain their natural order. HyTGraph recomputes the loaded subgraph only once because most updates can only pass two hops effectively [36]. Another benefit of this hub-vertex gathered method is that the vertices having a high probability of being activated (with large in-degree) are stored together. This property can help improve the effect of cost-aware task generation.

It is worth mentioning that the hub sorting does not need to be performed in each run. As long as performing the hub-sorting once in the data preparation stage, all the subsequent executions (of different algorithms) can benefit from it.
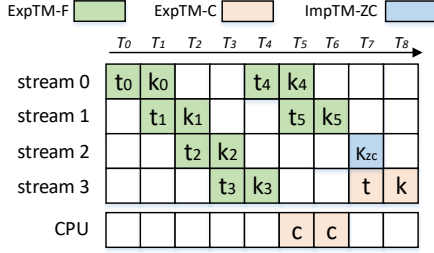
Fig. 6: An illustrative example of multi-stream scheduling, the "t" represents the transfer operation, "k" represents the computation kernel, and "c" represents the CPU-based compaction.

Δ**-driven priority scheduling.** For some iterative graph algorithms based on value accumulation, e.g., Δ-based PageRank and PHP algorithm [41], the contribution of vertices is directly reflected in their delta values (the messages to-be-accumulated). Prioritizing the vertices with large Δ value can help the downstream vertices accumulate updates more effectively [37], [39]–[41]. Since the original Δ-driven priority scheduling is vertex-centric, it can not be directly used in GPU-accelerated graph processing. To address this problem, HyTGraph implements Δ-driven scheduling with minor modifications. In each iteration, HyTGraph computes Δ value for all partitions and prioritizes those with large delta values. Similar to that of hub-vertex-driven priority scheduling, in Δ-driven scheduling, HyTGraph process the loaded partition only one more time.

B. 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. Figure 6 shows an illustrative example. 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-A) 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. After finishing all the computing tasks, HyTGraph will call Algorithm 1 to prepare for the next iteration.

C. Other Implementations

Implementation of processing kernels. HyTGraph provides three processing kernels for implementing ExpTM-filter, ExpTM-compaction, and ImpTM-zero-copy hybrid execution. Since the ExpTM-based engine needs to perform computation on partitioned subgraphs, we implement the processing kernels of ExpTM by extending SEP-Graph's processing

kernel to enable neighbor shifting on the edges-associated array [37]. While for ImpTM-zero-copy, HyTGraph uses the original kernel of SEP-Graph. HyTGraph inherits a series of inner-GPU optimizations from SEP-Graph, including data-/topology-driven switching [37] and Cooperative Thread Array (CTA) scheduling [20]. In addition, we implement bitmap-directed frontier optimization [38] to reduce the atomic conflict of active vertex maintenance.

Implementation of compaction. We implement a simple yet efficient parallel edge compaction engine by referring to Subway [33]. Since the physical locations of the edge-associated data are changed in the compaction stage, HyTGraph has to generate a new compressed neighbor index array and transfers it to the GPU along with the compacted edge array(s) for the ExpTM-compaction computation.

VII. EXPERIMENTAL EVALUATION

A. Experimental Setup

Environments. Our test platform is equipped with one Intel Silver 4210 2.20Ghz 10-core CPU, 128GB DRAM, and a NVIDIA GTX 2080Ti GPU with 34SMX clusters, 4352 cores, and 11GB GDDR6 global memory. The GPU is enabled with CUDA 10.1 runtime and 418.67 driver, the host side is running Ubuntu 18.04 with Linux kernel version 4.13.0. All the 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) [37]. We use both real-world graphs and synthesized graphs in our evaluation. The major parameters of graph datasets that are used in our experiments are presented in Table IV: Friendster-konect (FK) and friendster-snap (FS) are undirected social network datasets. Sk-2005 (SK) and uk-2007 (UK) are directed web graph datasets. Twitter (TW) is a directed social network dataset. The synthesized graphs used in our experiment are generated by RMAT [7] with the power-law distribution.

Systems for comparison. We compare HyTGraph with three representative and public available GPU-accelerated graph processing systems and a CPU-based graph processing system Galois [26] (Scaph [44] and Ascetic [35] are also available but we could not run them in our environment due to various CUDA errors, we were not able to resolve these errors after multiple email exchanges with the authors). Besides Subway [33] and EMOGI [25], Grus is a hybrid framework

TABLE IV: Dataset description.

Dataset	|V|	|E|	|E|/|V|	Size
sk-2005 [2] (SK)	50.6M	1.93B	38	28GB
twitter [1] (TW)	52.5M	1.96B	37	32GB
friendster-konect [1] (FK)	68.3M	2.59B	37	42GB
uk-2007 [2] (UK)	105.1M	3.31B	31	55GB
friendster-snap [3] (FS)	65.6M	3.61B	55	58GB
RMAT [7]	1-100M	0.1-6.4B	-	-

TABLE V: Comparison with other systems.

Alg.	System	Overall runtime (s)				
		SK	TW	FK	UK	FS
PR	Galois	21.3	66.3	293.6	28.5	342.4
	ExpTM-F	37.7	34.8	60.7	34.3	162.8
	ImpTM-UM	6.89	16.5	75.4	22.4	102.7
	Grus	**_1.72_**	12.2	52.2	14.8	79.8
	Subway	8.68	38.1	73.7	16.9	108.4
	EMOGI	18.6	21.4	51.1	12.4	68.3
	HyTGraph	2.85	**_11.5_**	**_30.1_**	**_4.71_**	**_40.8_**
SSSP	Galois	26.7	12.9	51.5	15.2	33.1
	ExpTM-F	60.9	15.1	50.4	60.9	70.1
	ImpTM-UM	12.7	10.1	37.2	18.6	34.9
	Grus	25.2	11.2	70.8	5.32	16.9
	Subway	14.6	10.9	20.8	18.4	27.7
	EMOGI	7.46	4.09	14.9	4.71	11.8
	HyTGraph	**_6.11_**	**_2.09_**	**_8.81_**	**_2.78_**	**_6.64_**
CC	Galois	23.9	15.7	35.9	55.1	39.4
	ExpTM-F	21.9	5.47	10.9	41.6	11.8
	ImpTM-UM	**_1.43_**	1.49	3.27	7.88	4.16
	Grus	2.09	1.36	3.21	5.17	4.69
	Subway	11.67	6.52	8.61	14.7	14.1
	EMOGI	4.01	1.96	2.71	4.54	3.76
	HyTGraph	3.65	**_1.19_**	**_2.01_**	3.86	**_2.59_**
BFS	Galois	16.2	7.55	12.5	15.2	14.7
	ExpTM-F	20.3	3.86	8.87	25.1	9.54
	ImpTM-UM	1.13	1.29	1.97	2.33	6.25
	Grus	**_0.83_**	1.11	1.85	2.37	3.35
	Subway	7.39	5.79	6.85	9.04	13.49
	EMOGI	1.06	1.04	**_1.44_**	1.26	**_1.97_**
	HyTGraph	0.93	**_0.85_**	1.82	**_0.88_**	2.54

Fig. 7: Execution path of HyTGraph and per-iteration runtime comparison with ExpTM-filter, EMOGI and Subway (FK).

[38] that combines ImpTM-unified-memory and ImpTM-zero-copy, when the storage space is large enough, it caches the transferred data in GPU through unified memory. When the device memory is full, Grus accesses the host data through zero-copy. Unlike HyTGraph, Grus' hybrid processing does not consider the processing overhead of the two approaches. In addition to these systems, we also implement pure ExpTM-filter and ImpTM-unified-memory in HyTGraph's codebase for a fair comparison. We use the default configuration of these systems and all the runtime results are measured by averaging the results of 5 runs.

B. Overall Performance

1) Comparison with ExpTM-F, Subway, and EMOGI: Table V shows the overall results. Due to the heavy redundant transfer, ExpTM-F shows worse performance than the others, the speedup of HyTGraph over ExpTM-F ranges from 2.01X (for PageRank on FK) to 28.52X (for BFS on UK) with an average of 8.99X. Neither Subway nor EMOGI is always better than the other. The speedup of HyTGraph over Subway ranges from 2.36X (for SSSP on FK) to 10.27X (for BFS on UK) with an average of 4.61X. 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
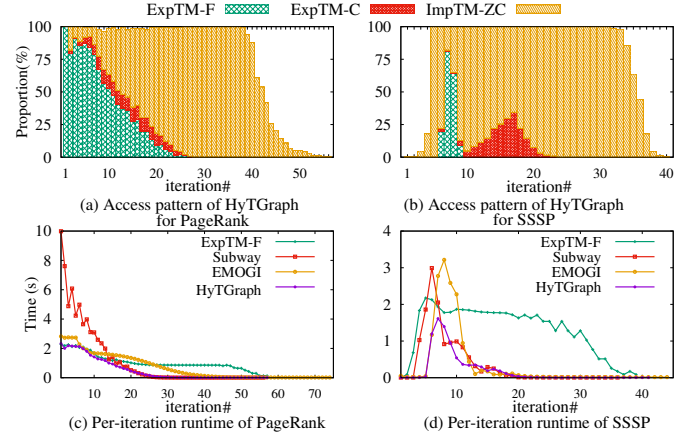
1.96X on average, with its speedups ranging from 1.10X to 6.53X. With the help of zero-copy, EMOGI achieves significant performance improvement on low-activeness subgraphs. While for the high-activeness subgraphs, especially those with dense and small degree vertices, EMOGI usually has low host-GPU utilization 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 merging overheads.

2) Comparison with Unified-Memory-based Approaches (ImpTM-UM and Grus): On SK graph, the unified-memory-based frameworks show better performance than the others for PageRank, CC, and BFS algorithms because the edge-associated data can be entirely cached in the GPU memory. UM-based approaches only transfer the data once. While, 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 13.14X and 2.37X speedups over ImpTM-UM and Grus, respectively.

3) Comparison with CPU-based Approach: From Table V, we can observe that the GPU-accelerated graph processing frameworks show significant performance improvement over CPU-based Galois. Specifically, HyTGraph shows on average 5.27x-12.78x speedups over Galois.

C. Execution Path Analysis

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 7 (a) shows the result of PageRank, the proportion of active partitions is high in the early iteration, and
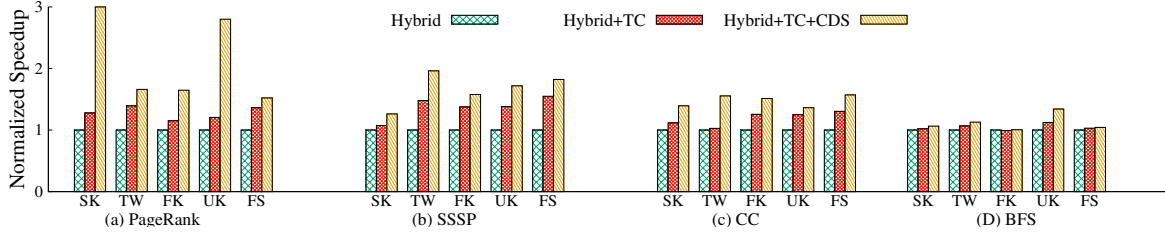
Fig. 8: Performance gain analysis of **T**ask **C**ombining (TC) and **C**ontribution-**D**riven **S**cheduling (CDS).

HyTGraph prefers ExpTM-filter. As the algorithm converges and many vertices become inactive, the proportion of ImpTM-zero-copy begins to increase. For SSSP in Figure 7 (b), there are few active vertices in the early and last few iterations, and 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 on some partitions.

Figure 7 (c) and (d) show the per-iteration runtime results 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 exactly the same. HyTGraph cannot consistently outperform the others in each iteration. While, through the hybrid transfer management, HyTGraph can achieve the minimum overall runtime.

D. Transfer Reduction Analysis

TABLE VI: Transfer reduction analysis.

Transfer volume / Edge volume					
Alg.	Dataset	ExpTM-F	Subway	EMOGI	HyTGraph
PR	SK	57.6X	2.46X	3.31X	**2.17X**
	TW	52.4X	**5.48X**	20.6X	10.9X
	FK	58.3X	**10.74X**	24.6X	12.01X
	UK	30.9X	1.79X	3.81X	**1.68X**
	FS	121.6X	**12.44X**	25.23X	12.62X
SSSP	SK	44.3X	4.23X	3.29X	**3.25X**
	TW	11.2X	2.07X	1.74X	**1.25X**
	FK	28.1X	**3.32X**	4.81X	4.60X
	UK	24.3X	1.78X	**1.11X**	1.13X
	FS	24.1X	3.19X	2.69X	**2.52X**

We analyze the effectiveness of HyTGraph's transfer reduction by comparing it with ExpTM-filter, Subway (ExpTM-compaction), and EMOGI (ImpTM-zero-copy). We run PageRank and SSSP on all the five real-world graphs and normalize the data transfer volume to the times of edge volume. Table VI shows the results, ExpTM-filter has the highest transfer volume. With the help of fine-grained zero-copy access, EMOGI achieves considerable transfer reduction. However, due to the lack of asynchronous scheduling, its transfer volume is still large. Benefiting from the CPU-based compaction, Subway is expected to have minimal data transfer volume. But the multi-round asynchronous processing performs differently on different algorithms. For PageRank algorithm based on value accumulation, the multi-round processing significantly reduces the transfer times because the

additional computations on partitioned subgraphs can still contribute to the final convergence. As processes the transferred subgraph only once more, HyTGraph has no transfer advantages over Subway for PageRank algorithm, especially on the small graph with few partitions, e.g., HyTGraph requires 2X data transfer compared to Subway on TW graph. HyTGraph achieves comparable data transfer volume with subway on SK graph (another small graph) because it benefits a lot from the contribution-driven priority scheduling (As illustrated in Figure 8 (a), the contribution-driven scheduling shows significant performance improvement on the two web graphs, SK and UK.). For the value-replacement-based SSSP algorithm, simply processing the transferred subgraph multiple times may cause stale computation problem (Section VI), leading to more computations and data transfers. We can observe that Subway transfers more data than EMOGI on SK, TW, UK, and FS for SSSP algorithm. In contrast, with the help of hybrid transfer management and asynchronous task scheduling, HyTGraph achieves significant transfer reduction in all cases and alleviates the stale computation problem.

E. Performance Gain of Task Combining and Contribution-Driven Scheduling

To analyze the performance gain of task combining and contribution-driven scheduling, we start from the pure hybrid transfer management with basic optimization (multi-stream scheduling) and integrate task combining (as described in section V-B), and contribution-driven scheduling (as described in section VI-A) one by one. Figure 8 shows the normalized speedups. The task combining (TC) can bring Hybrid an on average 1.28X, 1.37X, 1.19X, and 1.05X speedups on PageRank, SSSP, CC, and BFS, respectively. The contribution-driven scheduling (CDS) can further bring 2.18X, 1.21X, 1.25X, and 1.06X speedups over the hybrid processing with TC. Finally, the two proposed designs can bring an overall 2.78X, 1.67X, 1.47X, and 1.16X speedups over the raw hybrid transfer management, respectively. PageRank algorithm benefits most because the proposed asynchronous processing can effectively accelerate the convergence by prioritizing the vertices with large rank values. In contrast, BFS rarely benefits from the two designs because the vertices are activated only once during the iterative processing.

F. Sensitivity Analysis

Varying graph sizes. We compare HyTGraph with Grus, Subway, and EMOGI under variable graph sizes and report the results in Figure 9. When the graph size is small, Grus shows better performance because the data only needs to be
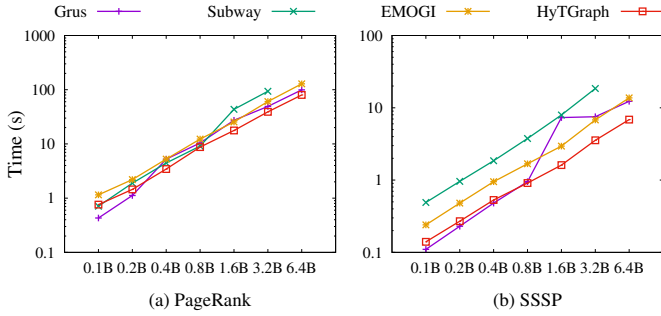
Fig. 9: Performance comparison with increasing graph size, the graphs are generated by RMAT with sizes from 0.1 Billion to 6.4 Billion (64X).
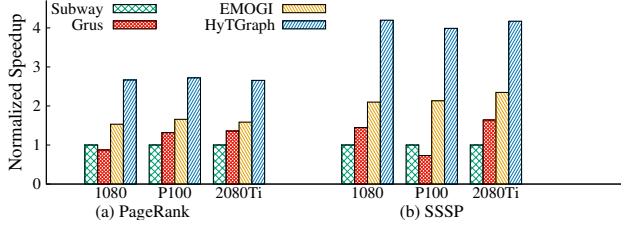

Fig. 10: Performance comparison on different GPUs (FS).

loaded once. While, as the graph size increases, the inefficient data transfer of unified-memory will reduce its performance. Subway fails to run the graph with 6.4B edges because of the integer overflow problem. As the graph size increases from 0.1B to 6.4B (64X), the runtimes of Grus, EMOGI, and HyTGraph for PageRank increase by 231.2X, 111.6X, and 105.39X, respectively. For SSSP algorithm, the runtime of Grus, EMOGI, and HyTGraph increase by 111.8X, 57.08X, and 49X, respectively. HyTGraph shows better performance when scaling to larger graphs.

Varying GPUs. We evaluate the performance of HyT-Graph on different GPUs, including GTX 1080 (2560cores, 8GB), TESLA P100 (3584cores, 16GB), and GTX 2080Ti (4352cores, 11GB) with FS graph. We normalize the runtimes of all systems to Subway and show the results in Figure 10. We can observe that HyTGraph outperforms the other three competitors. For PageRank, HyTGraph achieves 2.6X-2.7X, 2.0X-3.1X, and 1.6-1.7X speedups over Subway, Grus, and EMOGI, respectively. For SSSP, HyTGraph achieves 4.0X-4.2X, 2.5X-5.5X, and 1.7X-2.0X speedups over Subway, Grus, and EMOGI, respectively.

VIII. LIMITATIONS AND FUTURE WORK

Cost computation of ExpTM-C. The current version of HyTGraph uses an approximate method to compute the cost of ExpTM-C because the overhead of irregular main memory access is hard to quantify accurately. It would be interesting future work to model the ExpTM-C overhead through machine learning techniques.

Processing hyper-scale graph. For a hyper-scale graph whose vertex data exceeds a single GPU memory, processing it with GPU needs to partition the vertex data into smaller chunks that can fit into GPU memory. Such an approach requires frequent host-GPU vertex data swapping, leading to additional data transfer overhead. Therefore, designing new algorithms to optimize the host-GPU vertex data access and exploring whether the computation improvement can cover the additional I/O overhead are interesting and less studied problems. We will take them as our future work.

Adapting to GPU platforms with fast interconnects. Recently, the hardware makers have come up with fast interconnect technologies (e.g., NVIDIA NVlink [28] and Intel CXL [8]) to replace the slow PCIe bus, which can provide up to 900GB/s GPU-CPU interconnect bandwidth (NVlink-4.0 [28]). In a GPU-accelerated platform with fast interconnections, the main memory may become a new bottleneck of host-GPU data transfer [21]. We can improve HyTGraph by exploring the main memory access performances of different transfer methods and integrating the main memory accessing cost in our hybrid model to adapt to these new platforms.

IX. RELATED WORK

In-GPU-memory graph processing. To accelerate graph processing, the high parallelism of GPU has attracted great attention [10], [15]–[17], [24], [40], [43], [45]. Cusha [17] uses two novel data structures, named GShards and CW, to avoid non-coalesced memory access. Gunrock [40] performs computation on the frontier with data-centric abstraction. Tigr [32] proposes a virtual transformation to transform skewed graphs into virtual vertices for load-balancing. SEP-Graph [37] switches execution paths adaptively based on a selection in each of the three pairs of parameters, namely, Sync or Async, Push or Pull, and DD (data-driven) or TD (topology-driven).

Out-of-GPU-memory graph processing. GPU-accelerated graph processing has attracted extensive attention. Besides the systems mentioned above [11], [14], [25], [33]–[35], [38], [44], recent studies also propose CPU-GPU co-processing to accelerate large graphs computation [12], [22]. Totem [12] partitions a graph into two subgraphs, one for the CPU and one for GPU, keeping the number of data transfers to a minimum at the expense of severe load imbalance. Garaph [22] concurrently processes the active subgraphs on both the host and GPU. However, the CPU-based low-activeness subgraph processing may become a new bottleneck.

X. CONCLUSION

We present HyTGraph, a highly efficient GPU-accelerated graph processing framework by adaptively switching the transfer management approach 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. Our intensive experiments show the high effectiveness of HyTGraph.

References

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

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

[3] Stanford network analysis project. https://snap.stanford.edu/. 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 Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, editors, *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 H. C. Hunter, J. Moreno, J. S. Emer, and D. Sánchez, editors, *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 V. Sarkar and L. Rauchwerger, editors, *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 235–248. ACM, 2017.

[7] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, editors, *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 442–446. SIAM, 2004.

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

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

[10] A. Gaihre, Z. Wu, F. Yao, and H. Liu. XBFS: exploring runtime optimizations for breadth-first search on gpus. In J. B. Weissman, A. R. Butt, and E. Smirni, editors, *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.

[11] 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.

[12] 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 P. Yew, S. Cho, L. DeRose, and D. J. Lilja, editors, *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 345–354. ACM, 2012.

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

[14] 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.

[15] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, editors, *High Performance Computing - HiPC 2007, 14th International Conference, Goa, India, December 18-21, 2007, Proceedings*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer, 2007.

[16] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In C. Cascaval and P. Yew, editors, *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.

[17] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In B. Plale, M. Ripeanu, F. Cappello, and D. Xu, editors, *HPDC'14*, pages 239–252, 2014.

[18] 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.

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

[20] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 260–271. IEEE Computer Society, 2014.

[21] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Pump up the volume: Processing large data on gpus with fast interconnects. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1633–1649, 2020.

[22] 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 D. D. Silva and B. Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 195–207. USENIX Association, 2017.

[23] 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 A. K. Elmagarmid and D. Agrawal, editors, *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.

[24] 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.

[25] 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.

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

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

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

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

[30] NVIDIA. Nvidia v100 tensor core gpu. https://www.nvidia.com/en-us/data-center/v100/, 2022.

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

[32] A. H. N. Sabet, J. Qiu, and Z. Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, editors, *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.

[33] A. H. N. Sabet, Z. Zhao, and R. Gupta. Subway: minimizing data transfer during out-of-gpu-memory graph processing. In A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 12:1–12:16. ACM, 2020.

[34] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. 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 28:1–28:12. ACM, 2015.

[35] 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.

[36] K. Vora. LUMOS: dependency-driven disk-based graph processing. In D. Malkhi and D. Tsafrir, editors, *USENIX ATC 2019*, pages 429–442. USENIX Association, 2019.

[37] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang. Sepgraph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In J. K. Hollingsworth and I. Keidar, editors, *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 38–52. ACM, 2019.

[38] 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.

[39] 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 D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2439–2454. ACM, 2020.

[40] 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 R. Asenjo and T. Harris, editors, *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pages 239–252. ACM, 2016.

[41] 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.

[42] 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.

[43] 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 I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, editors, *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.

[44] 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 A. Gavrilovska and E. Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 573–588. USENIX Association, 2020.

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

[46] X. Zhu, W. Chen, W. Zheng, and X. Ma. In K. Keeton and T. Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 301–316. USENIX Association, 2016.

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