Received 19 February 2024, accepted 2 March 2024, date of publication 7 March 2024, date of current version 22 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3374513

RESEARCH ARTICLE

Graph Processing Scheme Using GPU With Value-Driven Differential Scheduling

SANGHO SONG[1], HYEONBYEONG LEE[1], YUNA KIM[2], JONGTAE LIM[1], DOJIN CHOI[3], KYOUNGSOO BOK[4], AND JAESOO YOO[1]

[1]Department of Information and Communication Engineering, Chungbuk National University, Seowon-gu, Cheongju-si, Chungcheongbuk-do 28644, South Korea
[2]Department of Big Data, Chungbuk National University, Seowon-gu, Cheongju-si, Chungcheongbuk-do 28644, South Korea
[3]Department of Computer Engineering, Changwon National University, Uichang-gu, Changwon-si, Gyeongsangnam-do 51140, South Korea
[4]Department of Software Convergence, Wonkwang University, Iksan-si, Jeollabuk-do 54538, South Korea

Corresponding author: Jaesoo Yoo (yjs@cbnu.ac.kr)

This work was supported by Chungbuk National University Korea National University Development Project, in 2022.

ABSTRACT Researchers have recently been using GPUs to process large quantities of graph data. However, the challenges in Host–GPU data transfer must be addressed to effectively use GPUs for graph processing. Although existing frameworks have attempted to mitigate this problem by managing active graph data transfers, issues persist owing to the need to divide graphs into subgraphs for parallel processing across multiple GPU cores. This division often leads to duplicated data transfers, resulting in high transmission overhead and low bandwidth utilization. To address these challenges and expedite graph computation, this study proposes a graph processing scheme using a GPU with value-driven differential scheduling. This approach involves dividing large graphs into subgraphs of similar sizes and contiguous vertices, allowing efficient parallelization on the GPU. The value of each subgraph is assessed based on its activity level, and its computation load is estimated using a differential subgraph scheduling technique. The proposed scheme distinguishes between high-value and low-value subgraphs and allocates them to different graph processing engines. This reduces the redundant data transmissions and enhances the transmission rate of active edges, thereby reducing the Host–GPU data transmission overhead. Experimental results demonstrate that the proposed scheme achieves a notable speedup of up to 6.6 times compared to the existing GPU-accelerated graph processing systems, including GraphCage and Subway.

INDEX TERMS GPU, graph processing, graph partitioning.

I. INTRODUCTION

Real-world applications of graph processing include pathfinding, social network analysis, and machine learning algorithms [1], [2], [3], [4], [5]. GPU-leveraging graph processing schemes are particularly promising for large parallel systems as they provide outstanding performance for graph algorithms that involve repetitive operations [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. A GPU, originally designed for graphics processing, inherently excels at executing several parallel operations. This parallelism is facilitated by the single instruction multiple data (SIMD) architecture inherent to GPUs. In SIMD computing, several values are processed simultaneously using a single instruction, contrasting with the typical structure of CPUs. Consequently, GPUs necessitate specific data structure and scheduling approaches to fully exploit their parallel capabilities [12], [18], [19], [20], [21], [22], [23]. Owing to their relatively smaller memory than the Host, GPUs can only accommodate a subset of the complete graph as input. When the size of the input graphs exceeds the GPU memory capacity, existing GPU-based systems encounter limitations [11], [15], [16], [20], [24], [25], [26], [27], [28], [29], [30], [31], [32]. This challenge becomes increasingly pronounced as the graph sizes in social networks expand alongside advancements in the network technology. Notably, data from platforms like Facebook now scale into the terabyte range, prompting researchers

The associate editor coordinating the review of this manuscript and approving it for publication was Massimo Cafaro.

to turn to GPUs for processing such massive graph data [4], [5].

Recently, there has been a notable focus on providing GPU-enabled graph processing while utilizing the Host memory for high-performance GPU graph processing and storage of large graphs. Achieving low bandwidth utilization is crucial for GPU graph processing, given the significant data transfer overhead between the Host and GPU memory. The bandwidth of the Host–GPU, typically via a PCIe interface, is approximately ten times slower than the bandwidth of the GPU global memory. In graph algorithms designed for GPUs, especially those dealing with structured graphs, accessing several vertices simultaneously is common, leading to challenges such as low work efficiency, high synchronization cost, and suboptimal data adjacency [6]. Excessively large graphs exacerbate issues such as cache misses and high memory access latency, particularly in graph algorithms with frequent graph input and output operations. Conversely, an overly small subgraph size leads to a proliferation of subgraphs, causing the overhead of merging partial results. Cache blocking, a well-established optimization strategy, addresses these challenges by partitioning the graph to enhance adjacency [3].

Graph partitioning is a fundamental technique used to process large graphs on GPUs with limited memory. Typically, graph processing algorithms involve iteratively sending messages from source vertices to other vertices. As this process iterates, the number of messages to be processed increases exponentially until the values of the receiving vertices no longer update, indicating convergence. This iterative process continues until the values of the vertices converge. In graph processing, active vertices refer to those that undergo updates during the iterative procedure, whereas inactive vertices remain unchanged. The majority of graph processing algorithms iterate until no active vertices remain [1]. This approach typically considers vertex-centered graph processing, where the algorithm iteratively operates on the graph at the vertex level. During each iteration, the algorithm focuses solely on active vertices, updating their adjacent edges and vertices accordingly. This process continues iteratively until no active data remain. Subsequently, previously processed subgraphs undergo reprocessing in the subsequent iteration.

However, due to the limited memory of GPUs, the existing method involves transmitting the entire subgraph of active vertices to the GPU, which not only wastes the Host–GPU bandwidth but also leads to duplicate data transfers. In this context, the Host refers to the system responsible for running the GPU and controlling the RAM data, while the GPU manages the global memory data.

Recent studies have focused on scheduling subgraphs for reuse, allowing them to be computed numerous times before being erased from the GPU memory [7]. Graphcage, for instance, divides data structures into smaller blocks and optimizes each block to fit into the cache, thereby accelerating graph operations by reordering memory accesses during computation [6]. However, Graphcage attempts to store all subgraphs in the limited GPU memory, resulting in challenges when integrating these subgraphs. Moreover, relying solely on cache blocking may not sufficiently reduce data transmission, especially given the high Host–GPU transfer time, which dominates the computation time of graph processing algorithms using GPUs. In contrast, Scaph divides a graph into subgraphs and calculates their values to schedule iterative operations on them [7], using a greedy vertex-cut approach [1]. It organizes the subgraph data in GPU memory into categories such as useful data (UD), potentially useful data (PUD), and never-used data (NUD) across current and future iterations. However, Scaph overlooks a graph partitioning scheme tailored to the GPU structure. Therefore, a processing technique is needed that considers the memory constraints and parallelized structures of GPUs, effectively dividing the graph and transmitting each subgraph independently.

In this paper, we propose a graph processing scheme using GPUs with value-driven differential scheduling. Our proposed scheme employs a subgraph partitioning algorithm tailored to the SIMD architecture of the GPU. Through experiments on load balancing, we dynamically divide the partitioned subgraph into appropriately sized graphs. To efficiently handle frequently utilized subgraphs in real-world graph processing, we introduce a differential subgraph processing scheme depending on their contribution to the overall graph. Subgraphs that are frequently accessed are identified as high value subgraphs, whereas those with lower usage frequency are categorized as low-value subgraphs. High-value subgraphs are transmitted to the GPU, while subgraphs likely to be reused in the future are stored in the GPU memory. By prolonging the storage of high-value subgraphs on the GPU, we aim to reduce redundant transfers. The optimal subgraph size is determined by comparing the number of subgraphs generated and the processing time of the graph algorithm using our partitioning scheme. The contributions of our proposed scheme are as follows.

- We propose a subgraph partitioning scheme and a subgraph management method that consider the Host–GPU transfer rate.
- The subgraph partitioning scheme considers the connectivity of the subgraphs and divides them based on the maximum number of vertices.
- It present a GPU-accelerated graph processing scheme called VDS, designed to optimize task scheduling and minimizes the Host–GPU duplicate transfers. This scheme is built upon our subgraph transfer management method, which considers the features of high and low-value subgraphs to enhance overall performance.

In this study, the performance is assessed using real-world graphs. The experimental results demonstrate a significant improvement, with a speedup of up to 10.7x compared to the state-of-the-art system, GraphCage [6], averaging 7.25x. Furthermore, compared to Subway [9], the speedup reaches up to 5.73x with an average improvement of 1.61x.

This paper is organized as follows: Section II examines related literature and outlines the problem statement. Section III elaborates on the proposed GPU-based graph algorithm processing scheme, detailing its process and features. Section IV demonstrates the superiority of the proposed scheme through system environment setup and performance evaluation. Finally, Section V presents the conclusion and outlines avenues for further studies.

II. RELATED WORK

Graph processing algorithms using conventional GPUs have been the focal point of recent research [12], [18], [19], [20], [21], [22]. Studies have delved into graph partitioning techniques and schemes for graph processing tailored to GPU architectures.

A. EXISTING GRAPH PARTITIONING SCHEMES

Preprocessing becomes necessary when performing operations that entail dividing the graph into subgraphs or transferring data for utilization on GPUs with limited memory capacity. Various graph partitioning schemes such as cache-blocking partitioning, greedy vertex-cut, and digraph have been explored [1], [6], [12]. However, conventional cache-blocking schemes, initially designed for CPU architecture, prove inefficient on GPUs. To address this, a cache-blocking partitioning scheme tailored for the GPU's L2 cache has been proposed, aiming to mitigate cache utilization deterioration resulting from inefficient memory accesses [6]. Partitioning the existing graph data into subgraphs based on vertices can reduce synchronization time between the cache and the global memory, thereby decreasing processing time. After processing each subgraph, the partial results are stored and aggregated. Subgraphs are divided to ensure that the vertex values of each subgraph can be accommodated in the cache. To aggregate partial results, a mapping of global and local IDs of vertices is maintained to construct an array of partial results. Rather than directly computing the sum, the partial result of each subgraph is stored in a partial result array, and all partial result arrays are aggregated after processing all subgraphs to obtain the final sum accurately.

The greedy vertex-cut algorithm [1] is a graph partitioning scheme that selects the vertex with the highest weight and divides the graph into two subgraphs based on this vertex. Although it does not guarantee an optimal partition at each step, it is highly efficient in most scenarios.

Digraph [12] employs a depth first search (DFS) technique to partition a subgraph into a set of edges starting from a designated starting vertex. The path generation process typically comprises two steps: Firstly, one root vertex is selected from each partition. Secondly, the graph is explored using DFS initiated from these root vertices to generate paths. Path lengths are often limited to be evenly distributed, achieved by setting a maximum search depth (i.e., the maximum number of vertices visited in DFS). This approach facilitates graph partitioning, which is particularly useful in distributed computing scenarios. Each partition can be allocated to a different server or CPU, and the uniform distribution of computational load across all partitions is ensured by controlling the path length. Furthermore, each path is transformed into a directed acyclic graph (DAG) sketch, allowing the utilization of DAG-based algorithms.

B. EXISTING GRAPH PROCESSING SCHEMES

To minimize the Host–GPU data transfers, novel graph processing schemes employing differential partitioning have been proposed [7], [17]. Scaph [7] adopts a strategy where subgraphs are constructed through greedy vertex-cuts. During computation, these subgraphs are classified and dispatched to two graph processing engines for one-hop iterations, managed through value-driven differential scheduling. The graph calculation continues until convergence is achieved. In Fig. 1, three forms of edge data based on repeatedly generated active vertices are illustrated. UD represents edge data adjacent to active vertices of a subgraph, such as V1→V3, V1→V4, and V2→V4 in Fig. 1. All edges originating from V1 and V2 are considered UD because V1 and V2 are active vertices. In the current iteration, UDs must be processed and sent to the GPU.

The edge data associated with all UD emerging from active vertices in the current iteration of the subgraph are termed PUD. In Fig. 1, V4 is a vertex connected to both active vertices V1 and V2. A V4→V5 transition exemplifies PUD, which, unlike UD, represents data not used in the current iteration but potentially essential for future calculations.

Edges associated with a vertex that converges and never becomes active are termed NUD. NUD represents edge data that will not be used in future computations.

The subgraph can assess the processing workload by measuring the edge data used in a one-hop unit iteration. This estimated workload can be expressed by the value of the subgraph. Subsequently, subgraphs are assigned to two distinct processing engines based on their workload values.

The filter method is used for high-value subgraphs, which involves sending the entire subgraph to the GPU for computation multiple times before being cleared from GPU memory. To minimize Host–GPU transfers, low-value subgraphs use the compaction method, extracting only UD data within the subgraph. With the filter method, the entire subgraph containing active vertices is sent to the GPU, where it undergoes multiple computations before being erased from the GPU memory. This method often results in high volumes of duplicate transfers, especially when transmitting active vertices. However, in cases where the partition is predominantly comprised of active vertices, it can effectively utilize PCIe bandwidth to its fullest extent. Meanwhile, the compaction approach significantly reduces transfers to the GPU by extracting only UD data within the subgraph. Nonetheless, it introduces compression overhead proportional to the ratio of active edges. It is worth noting that the compression step of compaction consumes approximately 34.5% of the total runtime [9]. Zero Copy utilizes the Host memory pinned to the GPU address space [8] to directly access the Host mem-

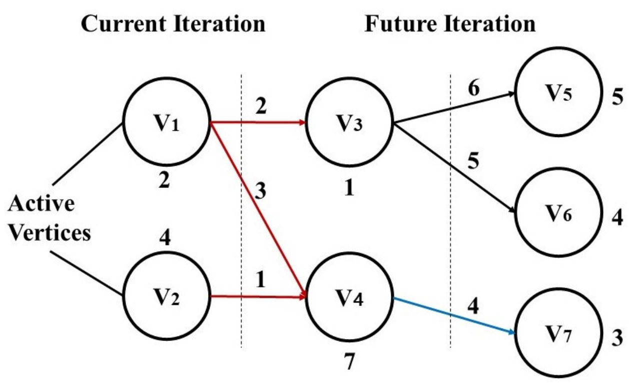


FIGURE 1. Three types of edge data.

ory using PCIe transaction layer packets (TLPs). Each TLP can simultaneously handle up to 256 memory requests, with each request capable of transferring 32/64/96/128 bytes of data, depending on the accessible data volume. Consequently, Zero Copy can simultaneously access edges of multiple distributed active vertices, with each occupying one or more memory requests. Zero Copy boosts low transfer overhead as it bypasses additional page access overhead. However, it is important to note that Zero Copy does not support data reuse, resulting in retransmission of the same data upon reuse.

C. ANALYSIS OF THE EXISTING SCHEMES

GraphCage successfully reduced cache utilization degradation by employing cache blocking techniques during subgraphs partitioning. However, the scheme's method of maintaining partial result arrays without directly recording the sum of partial operations in the subgraphs led to increased GPU memory usage and necessitated additional operations to obtain the sum of partial operations. In contrast, Scaph increased the Host–GPU bandwidth usage by calculating data necessary for iterative processes and scheduling value-oriented subgraph activities. However, it overlooked the parallel structure of the GPU and partitioned the graph using a graph partitioning scheme in its graph portioning scheme. Furthermore, preprocessing and sending low-value subgraphs to the GPU incurred labor overhead that required rebalancing. To address these limitations, our study introduces a subgraph partitioning scheme and a differential subgraph processing scheme. The subgraph partitioning scheme divides the subgraph into independent units suitable for GPU computation, ensuring an even combination of subgraph sizes while considering the parallel processing structure of the GPU and the connectivity of the partitioned subgraphs. These combined subgraphs are then categorized into high- and low-value subgraphs according to the frequency of active vertices. Each subgraph is managed separately by storing it in a worklist. High-value subgraphs are processed using filter and compaction techniques within an explicit transfer framework, whereas Zero Copy is used for processing low-value subgraphs within an implicit transfer framework.

III. PROPOSED GRAPH PROCESSING SCHEME

A. OVERALL STRUCTURE

Efficient graph partitioning and processing schemes compatible with GPUs with limited memory are essential. This study proposes an efficient graph processing scheme using GPUs, aiming to address challenges inherent in existing approaches. By partitioning the graph and predicting subgraph values using an optimized graph partitioning scheme, the proposed method minimizes processing time for repeated graph data while optimizing Host–GPU transmissions.

Fig. 2 shows the complete system architecture of our proposed scheme. It comprises three main modules: the graph partitioner, dispatcher, and value-driven subgraph process engine. The graph partitioner module divides the graph into subgraphs. The dispatcher module computes the subgraph values and routes them to different processing engines based on their values.

The goal of the graph partitioner module is to enable vertex-independent vertex processing on GPUs for parallel computation. It achieves this by considering the maximum number of vertices per subgraph and evenly dividing them to distribute the workload among the GPU cores. To mitigate the Host–GPU transfer time, which constitutes a significant portion of the calculation time for graph processing algorithms using GPUs, the dispatcher module distinguishes between subgraphs that require transfer and those that do not. The differential subgraph partitioning module comprises three components: Filter, Compaction, and Zero Copy. The filter module fully copies the subgraph to GPU memory. The compaction module compresses the subgraph based on active vertices and transfers them to GPU memory. The Zero Copy module processes subgraphs by handling independent memory requests for vertices and transferring active edges to the GPU. The distributed worklist undergoes modifications until the graph converges.

Fig. 3 shows the flowchart outlining the subgraph processing scheme. Upon receiving a graph as input, the graph partitioner module divides it into subgraphs. Subsequently, the dispatcher module evaluates the values of these subgraphs, proceeding to explore the next subgraph for computation. These subgraphs are then directed to the appropriate differential subgraph processing engine based on their values. The results from these processing engines are iteratively synchronized. Upon completion of vertex transversal, the results of the graph processing algorithm are produced as output.

B. GRAPH PARTITIONING

Recent advancements in handling massive graphs that surpass the capacity of GPU memory have been proposed to facilitate large-scale graph processing utilizing GPUs [9], [10], [13], [16], [17]. These approaches typically involve dividing large graphs into several subgraphs or chunks of uniform size that can fit within the global memory of the GPU and are subsequently transferred to the GPU for processing. This approach

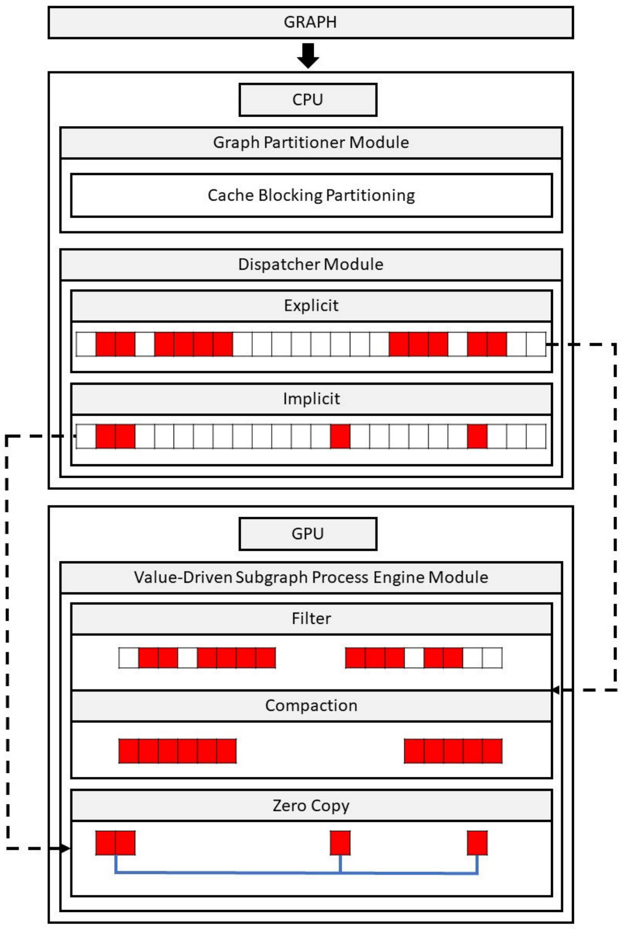


FIGURE 2. System architecture of the proposed scheme.

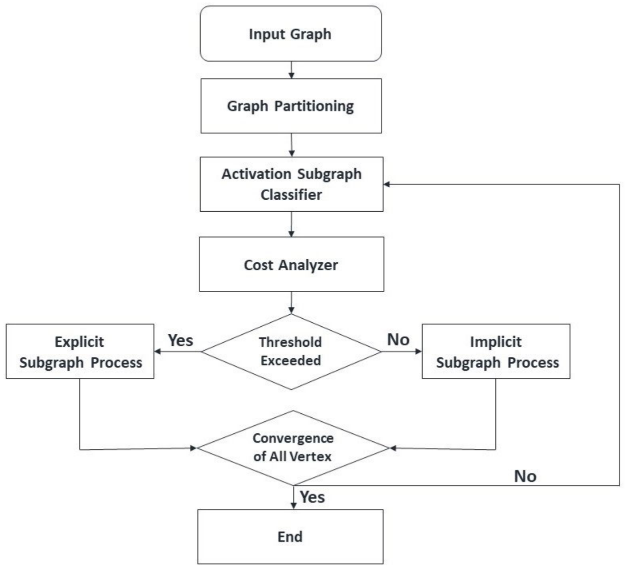

Input Graph
Graph Partitioning
Activation Subgraph Classifier
Cost Analyzer
Threshold Exceeded
Yes
No
Explicit Subgraph Process
Implicit Subgraph Process
Convergence of All Vertex
No
Yes
End

FIGURE 3. The overall graph processing of the proposed scheme.

significantly reduces GPU data access costs, accelerates vertex state propagation, and enhances overall GPU utilization.

The process of partitioning a graph for GPU parallel computing typically involves the following steps. Initially, a vertex from the graph is randomly selected. The graph is then divided based on the vertices connected to this selected vertex. Subsequently, the divided subgraph undergoes further partitioning, with consideration given to the vertex having the highest number of edges. This procedure is iteratively repeated until the subgraph reaches the desired size.

In recent GPU architectures, particularly focusing on the last level cache, or L2 cache, is crucial due to its role in optimizing latency and bandwidth between the LLC and the DRAM, which often serve as the bottleneck. To leverage this, static blocking is commonly employed, involving the partitioning of the graph into subgraphs before executing the algorithm. Alternatively, dynamic blocking uses intermediate buffers, dynamically distributing data across multiple buffers to enable each to enter the cache, thereby accumulating partial results. While dynamic blocking requires minimal alteration to the data structure, it can consume a significant amount of memory space due to the intermediate buffers. Moreover, the dynamic overhead incurred by adding data and reading data from these buffers must be considered. Given that graph algorithms typically involve a high number of iterations, static blocking is often preferred to mitigate dynamic overhead. This choice leads to a relatively large performance gain from preprocessing, facilitating the partitioning of the graph into subgraphs that operate efficiently on the GPU, thereby enhancing memory access efficiency.

Table 1 provides a summary of the edge data distribution in the graphs before and after partitioning, encompassing all data utilized in the performance evaluation. Prior to partitioning, it was observed that 79.1% of the subgraph vertices had fewer than nine edges, whereas 20% had more than nine edges. Following partitioning, over 90% of the subgraph vertices exhibited fewer than eight edges, with all vertices having fewer than 16 edges. This distribution reflects a balanced allocation of edges per vertex in the subgraph, indicative of effective load balancing.

Fig. 4 shows a partitioning scheme-generated subgraph. The greedy vertex-cut method adheres to three rules. Firstly, an edge must be allocated to the intersecting subgraph if it intersects subgraphs SG(s) and SG(d). Secondly, if only one subgraph of either vertex is assigned, the edge is allocated to that subgraph. Thirdly, if both vertices are indecisive, the edge is allocated to the subgraph with the fewest stored vertices. Greedy vertex-cut, shown in Fig. 4(a), considers adjacency in graphs during division. Consequently, the resulting subgraphs exhibit varying widths. Notably, certain subgraphs, like Subgraph 3 in Fig. 4(a), may be larger than others. To promote load balancing, the proposed scheme calculates the average size of partitioned subgraphs and assigns edges from larger subgraphs to smaller ones. This approach ensures an even distribution of edges among subgraphs. As shown in Fig. 4(b), partitioning subgraphs according to these principles

TABLE 1. Number of edges per vertex in a graph.

degree	Original Graph	Subgraphs
0 ~ 8	79.1%	90.7%
8 ~ 16	17.6%	10.3%
17~ 31	2.6%	0%
32 ~	0.7%	0%

yields evenly distributed subgraphs with adjacency between vertices and edges. By optimizing subgraph partitioning, the utilization of the L2 cache in the GPU can be enhanced, thereby accelerating internal GPU calculations. Furthermore, since the Host–GPU transfer time dominates the processing time of graph algorithms using GPUs, limiting the number of regularly transmitted subgraphs to the GPU by sequentially generating subgraphs can further optimize performance.

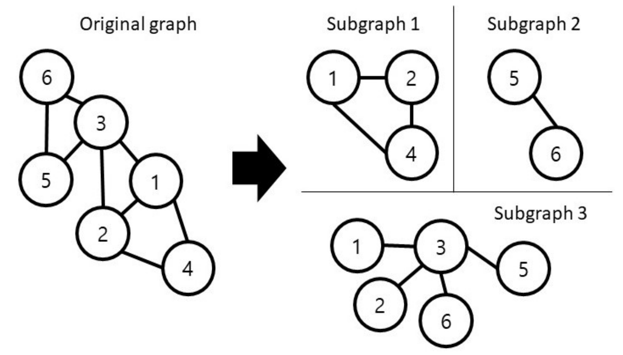
C. DIFFERENTIAL SUBGRAPH SCHEDULING

A comparison of the quantities of UD, PUD, and NUD within a subgraph, applied for a graph algorithm operating on Twitter, reveals that as iteration progress, more NUD are redundantly transmitted [33]. Existing graph processing schemes often inefficiently utilize Host–GPU bandwidth by repeatedly transferring data. Typically, there is significant redundant transmission of PUDs in the initial iteration, followed by redundant transmission of NUDs as iteration advances. Throughout iterations, the engagement of the same vertex can occur numerous times, resulting in dynamic variations in UD, PUD, and NUD. Considering PUDs alongside UDs can improve total graph processing time by preemptively sending PUDs to the GPU. Although not used in the current iteration, PUDs may be employed in subsequent iterations, thereby optimizing processing efficiency.

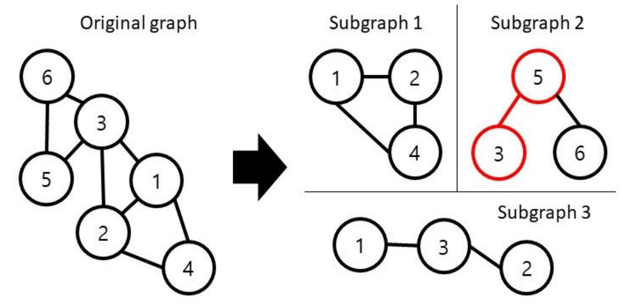
In Fig. 5, the iterations of the graph algorithm are depicted. During the first iteration, vertices 4 and 7 are activated, subsequently triggering vertices 1 and 6 in the second iteration. Notably, subgraph 2, having already been broadcasted, is omitted from transmission to save transmission time.

Table 2 presents the formula for calculating the overhead of each subgraph during processing in the study [14]. In the formulas, $D(v)$ denotes the number of edges connected with a vertex, d_1 denotes the memory occupancy of the vertex, m denotes the maximum request capacity of the memory, and MR denotes the maximum number of TLP requests in PCIe 3.0, indicating the compression throughput. Additionally, am denotes the additional transfer overhead for Zero Copy to access edge arrays with non-aligned vertices. The number of genuine TLPs is calculated as $d_1 / m / MR$. RTT refers to the round-trip time for TLP processing on PCIe.

The transmission overhead of the filter is computed as $D(v) * d / m / MR$. Compaction includes a CPU-based compression method, thus it includes both data transport and compression overhead. The transmission overhead is represented by $D(v) * d + |Ai| * d$, while the compression overhead is denoted by $D(v)\ d1 + |Ai\ |\ d2/T$. In the case of Zero Copy, each



(a) Greedy vertex-cut



(b) Proposed vertex-cut

FIGURE 4. Processing according to the partitioning scheme.

active vertex triggers a separate request. Consequently, the transmission overhead is calculated as $((D(v) * d / m) + am(v)) / MR$.

To schedule subgraphs efficiently, we calculate their data transmission costs and prioritize sending them in order of low transmission costs. Table 2 shows equations for the Filter, Compaction, and Zero Copy methods to compute graph processing overhead. The method with the most efficient transmission overhead, as determined by the equations, is selected. Specifically, if the transmission overhead exceeds 80% of the threshold, the Compaction method is chosen. For transmission overhead between 40% and 80%, the filter method is selected. If the overhead is less than 40%, the Zero Copy method is preferred. Filter and Compaction subgraphs primarily comprises UDs, with the majority of the subgraph transmitted to the GPU. In contrast, Zero Copy subgraphs are primarily composed of NUDs. Consequently, only UDs are extracted from the CPU and transmitted to the GPU for Filter and Compaction subgraphs, ensuring maximum utilization of edge data for calculation in both scenarios.

In scenarios where active edges are present within the subgraph, the filter method employs cudamemcpy to transmit the entire subgraph to the GPU. Conversely, compaction minimizes the volume of data transferred by using additional CPU-based compression, thereby transmitting only active edges to the GPU for processing. However, due to the additional CPU-based compression overhead, compaction is employed when its overhead exceeds 80% of that of the Fil-

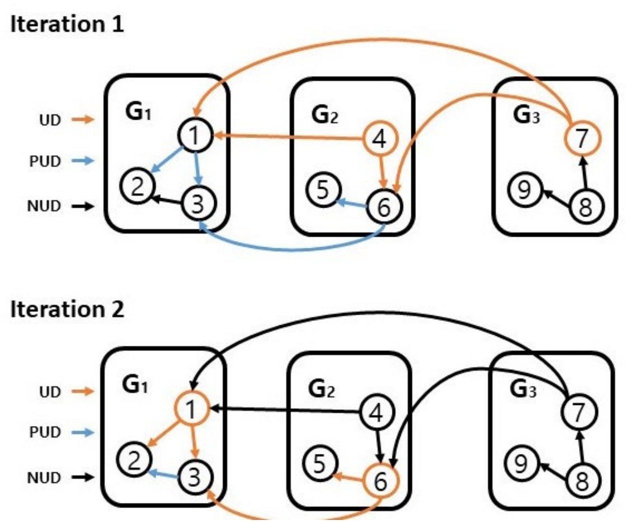


FIGURE 5. Iterative of a graph algorithm.

TABLE 2. Graph processing overhead.

Filter	$Tef_i = \left[\left(\sum_{v \in P_i} [\![D(v)) \times \frac{d}{m \times MR}\right]]\!] \times RTT$ *(1)*
Compaction	$Tec_i = [(\sum_{v \in A_i} D(v) \times d_1 + \lvert A_i \rvert \times d_2)/m/MR] \times RTT + \sum_{v \in A_i} D(v) \times d_1 + \lvert A_i \rvert \times \frac{d_2}{Thpt}$ *(2)*
Zero Copy	$Tec_i = (\sum_{v \in A_i} (\left[D(v) \times \frac{d_1}{m}\right] + am(v)))/MR] \times RTT$ *(3)*

ter method. For processing Zero Copy subgraphs containing unused NUDs, Zero Copy is utilized to handle each active vertex with an individual memory request.

Algorithm 1 depicts the subgraph scheduling algorithm employed in the proposed graph processing scheme. Initially, Graph G is partitioned into subgraphs G1 through Gn. Ensuring that Filter and Compaction subgraphs can effectively utilize UD and PUD edge data, regularly provided to the GPU, is crucial. Following comparison of the graph processing overhead, the Filter, Compaction, and Zero Copy modules are distinguished into worklists and transmitted accordingly. These three worklists operate independently and in parallel. In iterative graph processing, the proposed graph scheduling scheme maximizes the use of subgraph UDs and PUDs. Zero Copy subgraphs reduce the quantity of data exchanged between the Host and GPU by extracting the UDs and transmitting them to the GPU. At the conclusion of each iteration (line 15), updated vertices are transferred from the GPU to the CPU. Notably, only edges that have been updated are transmitted.

IV. PERFORMANCE EVALUATION

To showcase the superiority of our proposed graph partitioning scheme, we conducted a performance comparison

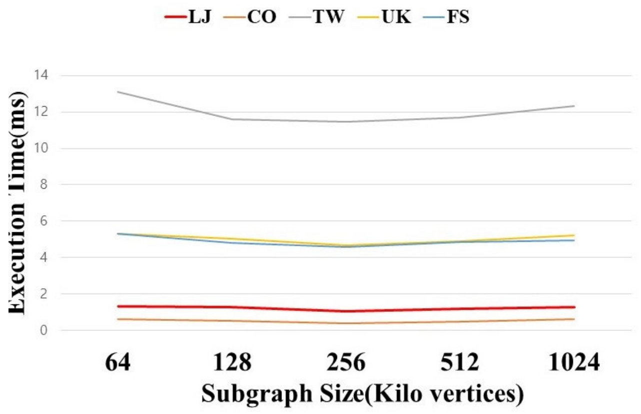


FIGURE 6. Graph algorithm processing time according to the subgraph size.

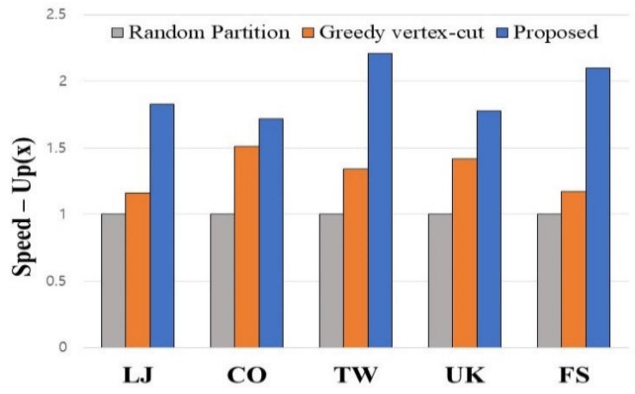


FIGURE 7. Graph algorithm processing time by partitioning scheme.

with existing strategies. Table 3 provides a summary of the performance evaluation environment. The Host system runs on a Linux-based Ubuntu operating system, powered by an Intel (R) Core (TM) i7-9700KF CPU @ 3.60 GHz 64-bit architecture, and equipped with 32 GB memory. The GPU utilized is a GeForce RTX 3060TI with 8 GB of GPU RAM. The graph data comprises a set of edge data.

Table 3 presents the datasets sourced from Stanford's extensive network dataset collection [34]. Our performance evaluations were conducted using the five real graph datasets shown in Table 3. These datasets include: LiveJournal (LJ), which includes data extracted from the online social network LiveJournal; Com-Orkut(CO), sourced from the online social network Orkut; UK-2005 (UK), directed web graph datasets; Twitter (TW), a directed social network dataset; and Friendster-snap (FS), an undirected social network dataset.

We explored the impact of varying subgraph sizes on the performance of our proposed system. The number of vertices per subgraph was adjusted within the range of 64 to 1024k during the partitioning process. Through our performance evaluations, we identified the optimal subgraph size. Fig. 6 depicts the processing time of the graph algorithm plotted against the number of subgraphs. The horizontal

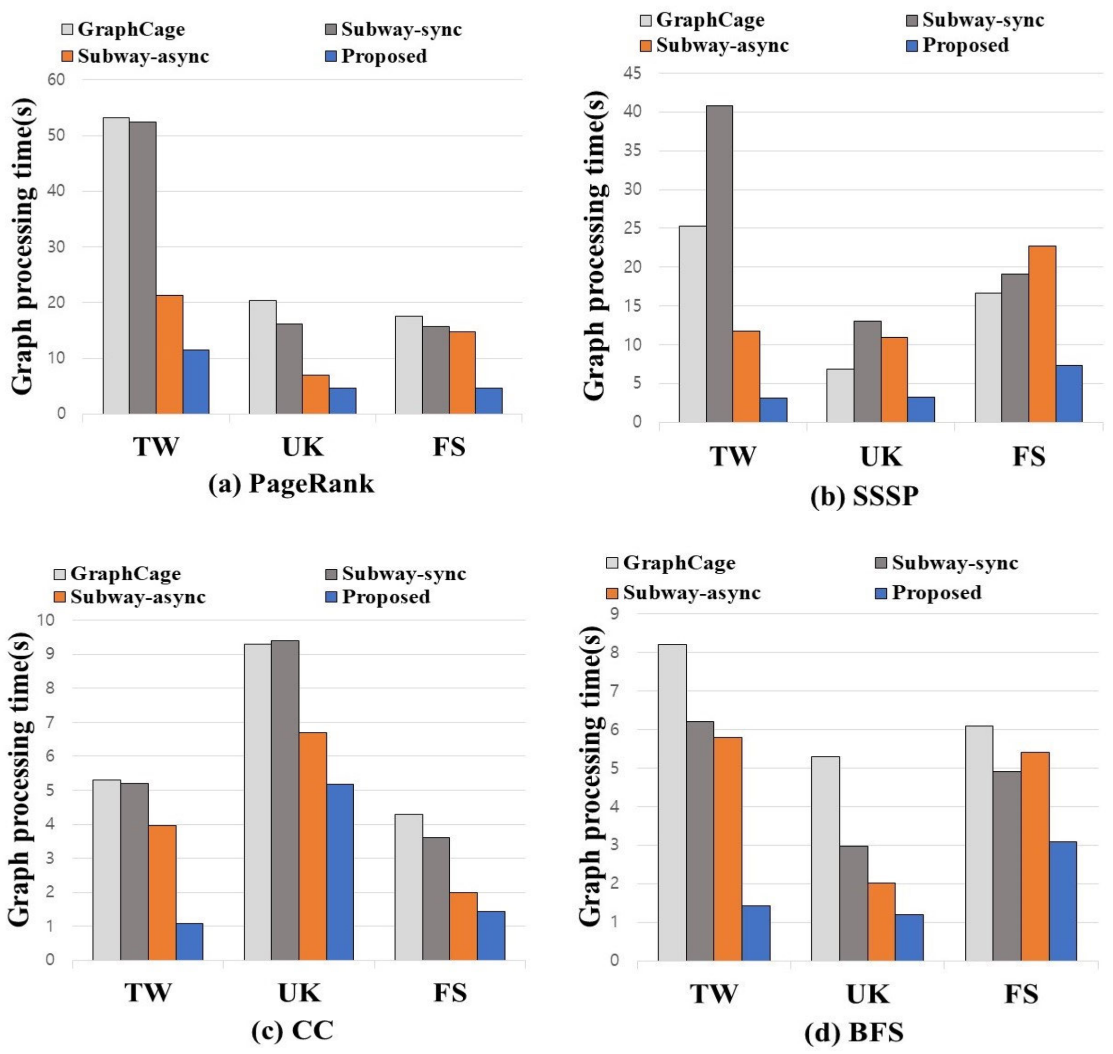


FIGURE 8. Comparison with other systems.

axis represents the size of the partitioned subgraph, whereas the vertical axis represents the percentage improvement in graph algorithm processing performance. For our evaluations, we utilized the PageRank algorithm and evaluated its performance across five real graph data sets. It is important to note that excessively small subgraphs may introduce overhead, while overly large subgraph sizes can compromise memory efficiency.

Fig. 7 illustrates a comparison of the processing times among graph algorithms employing graph partitioning schemes. The graph processing time for each dataset was determined by partitioning it using different partitioning algorithms and subsequently applying the proposed graph algorithm processing scheme. The horizontal axis represents the data set, while the vertical axis represents the improvement ratio with respect to the random partition processing time, which is set to 1. Following performance evaluation, the proposed scheme exhibited an average performance improvement of 1.93x and 1.46x compared to the Random Partition and Greedy vertex-cut division schemes, respectively. The Greedy vertex-cut scheme lacked load balance due to its emphasis on distributing adjacent vertices and edges to the same data. Conversely, the proposed partitioning scheme ensures good load balancing by evenly dividing the subgraph sizes. Furthermore, it distributes adjacent vertices and edge data to a single subgraph before partitioning it into equal subgraphs. Consequently, this approach improves the efficiency of concurrent processing on GPUs while reducing the access time of each subgraph. This performance evaluation demonstrates the superior performance of the proposed partitioning scheme.

Fig. 8 shows a comparison of the average processing time of the graph algorithms between the existing schemes and proposed scheme. The proposed scheme demonstrated substantial performance improvements of 4.27x, 3.82x, 2.2x, 6.6x, 3.4x for SSSP, 3.2x, 3.02x, 2.1x for CC, and 4.03x,

Algorithm 1 Proposed Graph Scheduling

```
1  Input: active vertex set {$G_1, \ldots, G_N$} of $N$ partitions,
2  Output: Transfer engine selection. Filter,
     Compaction, ZeroCopy
3  VertexInitialization (G);
4  Gactive ← FindActiveSubgraph (G);
5  Transfer VertexStatues from GPU to CPU;
6  while  Gactive != 0 do
7      if  Predictor(G) == 'Filter' then
8          Push(Fworklist, G);
9
       else if  Predictor(G) == 'Compaction' then
10         Push(Cworklist, G);
11
       else
12         Push(Zworklist, G);
13
       end
14     Gactive ← FindActiveSubgraph(G);
15     Transfer VertexStatues from GPU to CPU;
16
   end
```

TABLE 3. Graph datasets properties.

Dataset	V	E	E/V	Description
LiveJournal(LJ)	4.8M	68M	14.2	LiveJournal social network
Com-Orkut(CO)	3M	117M	38.5	Orkut social network
UK-2005(UK)	39M	0.93B	23.7	Websites in the UK
Twitter-2010(TW)	41M	1.46B	35.2	Twitter user connection
Friendster-snap(FS)	65.6M	3.61B	55	Friendster online social network

2.79x, and 2.48x for BFS, compared to GraphCage, Subway-sync, and Subway-async. In GPUs-based graph processing, a significant portion of the processing time is attributed to graph transmission. GraphCage suffers from transmitting redundant graph data, leading to inefficiencies. In contrast, Subway mitigates unnecessary data transmission between the GPU and CPU, thereby outperforming Graphcage. However, Subway-sync's performance is adversely affected by page faults during data transmission. For example, in Fig. 8, Subway-sync underperforms compared to Graphcage due to page faults in SSSP processing with twitter data. Additionally, Subway-async requires preprocessing time for data compression.

CPU-based compression and preprocessing currently constitute 30–40% of the overall runtime. The proposed scheme aims to enhance GPU utilization and optimize the Host–GPU transmission efficiency by implementing one of the following strategies: Compaction, Filter, and Zero Copy.

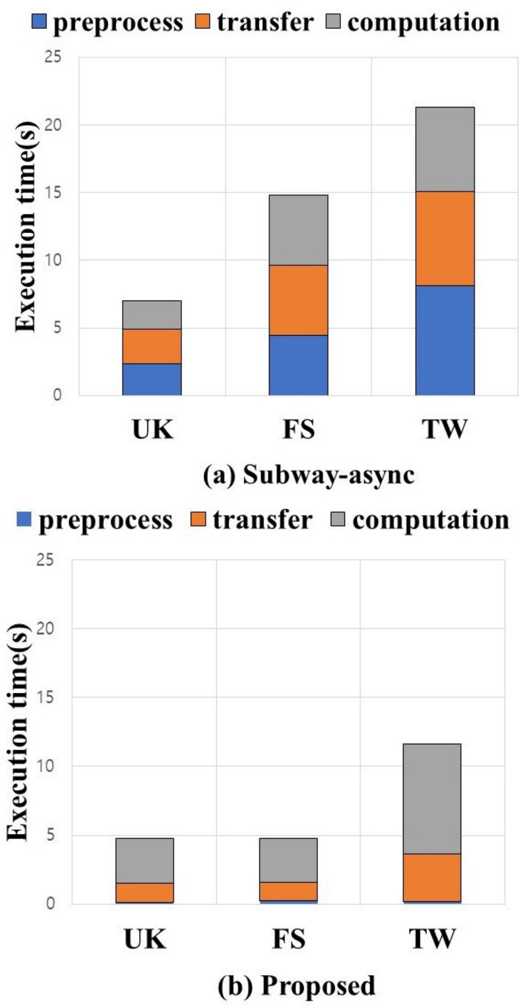


FIGURE 9. Execution time breakdown of PageRank.

Fig. 9 shows the breakdown of total processing times for Subway-async and the proposed technique during PageRank computation. In Fig. 9 (a), Subway-async accounts for 33% of the average total processing time for preprocessing, 35% for graph transmission, and 32% for data operations. Meanwhile, in Fig. 9 (b), the proposed scheme allocates an average of 5% of the total processing time to preprocessing, 35% to transmission, and 60% to data operations. The proposed scheme shows performance improvement by reducing redundant operations and eliminating unnecessary graph preprocessing time in graph data processing.

Due to limitations in the experimental setup, a thorough performance assessment could not be conducted. Scaling up the numbers of CPUs and GPUs might potentially enhance the efficiency of modules processed by the devices. Furthermore, integrating multiple CPUs and GPUs is anticipated to notably decrease communication costs between the CPU and GPU during graph algorithm processing.

V. CONCLUSION

In this paper, we propose a graph processing scheme using GPU with value-driven differential scheduling, facilitated by efficient subgraph partitioning. Our proposed approach aims to maximize GPU parallelism efficiency by effectively managing load balancing through a subgraph partitioning scheme that considers graph independence. Furthermore, we reduce graph transfers by minimizing duplicate graph transfers via a differential subgraph scheduling technique. Subgraphs are partitioned considering the GPU L2 cache and further divided into three graph processing engines to efficiently handle the transmission of active vertices and edges. Performance evaluation, comparing graph processing speed and transmission volume, demonstrates that processing speed increases with subgraph size and differential subgraph scheduling. Our method effectively handles recursive traversal graph algorithms like SSSP, BFS, PageRank, and CC using graph representation and processing tailored to the SIMD structure of the GPU. In the future, we plan to explore graph processing algorithms leveraging multiple GPUs for enhanced scalability and performance.

REFERENCES

[1] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2012, pp. 17–30.

[2] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2014, pp. 599–613.

[3] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large scale graph computation on just a PC," in *Proc. USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2012, pp. 31–46.

[4] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing—'ABSTRACT,'" in *Proc. 28th ACM Symp. Princ. Distrib. Comput.*, New York, NY, USA, Aug. 2010, pp. 135–145.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015.

[6] X. Chen, "GraphCage: Cache aware graph processing on GPUs," 2019, *arXiv:1904.02241*.

[7] L. Zheng, X. Li, Y. Zheng, Y. Huang, X. Liao, H. Jin, J. Xue, Z. Shao, and Q.-S. Hua, "Scaph: Scalable GPU-accelerated graph processing with value-driven differential scheduling," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2020, pp. 573–588.

[8] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W. M. Hwu, "EMOGI: Efficient memory-access for out-of-memory graph-traversal in GPUs," *Proc. VLDB Endowment*, vol. 14, no. 2, 2020, pp. 114-127.

[9] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-GPU-memory graph processing," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, p. 12.

[10] 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 *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2017, pp. 195–207.

[11] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs," in *Proc. 23rd Int. Symp. High-Performance Parallel Distrib. Comput.*, Jun. 2014, pp. 239–251.

[12] 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 *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 601–614.

[13] M. Wang, C.-C. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proc. 14th EuroSys Conf.* New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–17.

[14] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a GPU," in *Proc. 26th Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, Sep. 2017, pp. 233–245.

[15] A. H. N. Sabet, J. Qiu, and Z. Zhao, "TIGR: Transforming irregular graphs for GPU-friendly graph processing," in *Proc. ACM SIGPLAN Notices, Assoc. Comput. Machinery*, 2018, pp. 622–636.

[16] 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 *Proc. 24th Symp. Princ. Pract. Parallel Program.*, Feb. 2019, pp. 38–52.

[17] Q. Wang, X. Ai, Y. Zhang, J. Chen, and G. Yu, "HyTGraph: GPU-accelerated graph processing with hybrid transfer management," in *Proc. IEEE 39th Int. Conf. Data Eng. (ICDE)*, Los Alamitos, CA, USA, Apr. 2023, pp. 558–571.

[18] H. Zhu, L. He, S. Fu, R. Li, X. Han, Z. Fu, Y. Hu, and C.-T. Li, "WolfPath: Accelerating iterative traversing-based graph processing algorithms on GPU," *Int. J. Parallel Program.*, vol. 47, no. 4, pp. 644–667, Aug. 2019.

[19] V. Jatala, R. Dathathri, G. Gill, L. Hoang, V. K. Nandivada, and K. Pingali, "A study of graph analytics for massive datasets on distributed multi-GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2020, pp. 84–94.

[20] M. A. Awad, S. Ashkiani, S. D. Porumbescu, and J. D. Owens, "Dynamic graphs on the GPU," in *Proc. Int. Parallel Distrib. Process. Symp.*, May 2020, pp. 739–748.

[21] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "Faim-Graph: High performance management of fully-dynamic graphs under tight memory constraints on the GPU," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2018, pp. 754–766.

[22] L. Wan, W. Zheng, and X. Yuan, "Efficient inter-device task scheduling schemes for multi-device co-processing of data-parallel kernels on heterogeneous systems," *IEEE Access*, vol. 9, pp. 59968–59978, 2021.

[23] L. Wan, W. Zheng, and X. Yuan, "HCE: A runtime system for efficiently supporting heterogeneous cooperative execution," *IEEE Access*, vol. 9, pp. 147264–147279, 2021.

[24] D. H. Kim, R. Nagi, and D. Chen, "Thanos: High-performance CPU-GPU based balanced graph partitioning using cross-decomposition," in *Proc. 25th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2020, pp. 91–96.

[25] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "GPU-accelerated subgraph enumeration on partitioned graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 1067–1082.

[26] A. Barvinok and P. Soberón, "Computing the partition function for graph homomorphisms with multiplicities," *J. Combinat. Theory A*, vol. 137, pp. 1–26, Jan. 2016.

[27] B. Goodarzi, F. Khorasani, V. Sarkar, and D. Goswami, "High performance multilevel graph partitioning on GPU," in *Proc. Int. Conf. High Perform. Comput. Simul. (HPCS)*, Jul. 2019, pp. 769–778.

[28] R. Panja and S. S. Vadhiyar, "HyPar: A divide-and-conquer model for hybrid CPU–GPU graph processing," *J. Parallel Distrib. Comput.*, vol. 132, pp. 8–20, Oct. 2019.

[29] M. S. 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 *Proc. ACM SIGMOD Int. Conf. Manag. Data, Assoc. Comput. Machinery*, Jun. 2016, pp. 447–461.

[30] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proc. 12th Eur. Conf. Comput. Syst.*, Apr. 2017, pp. 527–543.

[31] 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 *Proc. 21st Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, Sep. 2012, pp. 345–354.

[32] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 265–266, Dec. 2015.

[33] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web*. New York, NY, USA: Association for Computing Machinery, 2010, pp. 591–600.

[34] J. Leskovec and A. Krevl. (Jun. 2014). *SNAP Datasets: Stanford Large Network Dataset Collection*. [Online]. Available: http://snap.stanford.edu/data

SANGHO SONG received the M.S. degree in information and communication engineering from Chungbuk National University, South Korea, in 2022. His research interests include big data processing, graph partitioning, and graph processing.

HYEONBYEONG LEE received the B.S. and M.S. degrees in computer engineering from Korea National University of Transportation, South Korea, in 2016 and 2018, respectively, and the Ph.D. degree in information and communication engineering from Chungbuk National University, South Korea, in 2023. His research interests include big data processing, indexing schemes, and distributed computing.

YUNA KIM received the M.S. degree from the Department of Big Data, Chungbuk National University, South Korea, in 2022. Her research interests include location-based services, big data processing, machine learning, and social network services.

JONGTAE LIM received the B.S., M.S., and Ph.D. degrees in information and communication engineering from Chungbuk National University, South Korea, in 2009, 2011, and 2015, respectively. He is currently a Research Professor with Chungbuk National University. His research interests include moving object databases, spatial databases, location-based services, P2P networks, and big data.

DOJIN CHOI received the B.S. and M.S. degrees in computer engineering from Korea National University of Transportation, South Korea, in 2014 and 2016, respectively, and the Ph.D. degree in information and communication engineering from Chungbuk National University, South Korea, in 2020. He is currently an Assistant Professor of computer engineering with Changwon National University. His research interests include location-based services, big data processing, continuous query processing, and distributed computing.

KYOUNGSOO BOK received the B.S. degree in mathematics and the M.S. and Ph.D. degrees in information and communication engineering from Chungbuk National University, South Korea, in 1998, 2000, and 2005, respectively. He is currently an Assistant Professor of software convergence technology with Wonkwang University, South Korea. His research interests include database systems, location-based services, mobile ad-hoc networks, big data processing, and social network services.

JAESOO YOO received the M.S. and Ph.D. degrees in computer science from Korea Advanced Institute of Science and Technology, South Korea, in 1991 and 1995, respectively. He is currently a Professor of information and communication engineering with Chungbuk National University. His research interests include database systems, storage management systems, sensor networks, distributed computing, big data processing, and social network services.