Optimization of Large-Scale Graph Processing with Advanced Partitioning and Thread Grouping

Muhammad Numan, Abdul Haseeb Malik, Qazi Ejaz Ali, Waheed ur Rehman, Muhammad Haseeb.
University of Peshawar.
*Correspondence: m.numan6082024@uop.edu.pk

Citation | Numan. M, Malik. A. H, Ali. Q. E, Rehman. W. U, Haseeb. M, "Optimization of Large-Scale Graph Processing with Advanced Partitioning and Thread Grouping", IJIST, Vol. 8 Issue. 4 pp 1639-1659, July 2026
DOI | https://doi.org/10.33411/IJIST/1944
Received | June 12, 2026 **Revised |** July 10, 2026 **Accepted |** July 13, 2026 **Published |** July 18, 2026.

Large-scale graphs are a natural way of modeling entities with complex relationships. Irregular structures and relationships complicate the processing of these large-scale graphs. Partitioning enables efficient processing of these graphs; however, it may result in information loss and imbalanced workload distribution. The processing of large graphs is greatly limited by these challenges as they make it difficult to scale and efficiently process them. This paper provides a large-scale graph processing framework that involves graph partitioning and thread grouping to enhance memory usage and computational efficiency of a GPU-based system. The proposed partitioning algorithm creates subgraphs with minimum cross-partitions and minimum cross-partition edges. A thread grouping algorithm efficiently processes these partitions in parallel, minimizing inter-partition connectivity. It creates an execution pack of partitions that can be efficiently processed on a GPU at runtime based on available memory, while allowing hardware resources to be used elastically. The GraphSAGE architecture is used to evaluate the framework with several real-world benchmark datasets, such as OGBN-Products, Reddit, and LiveJournal. The experimental findings indicate that the proposed execution strategy achieves a substantial decrease in memory pressure and preserves competitive classification accuracy. The proposed work provides better stability and scalability of training without compromising the performance of the model, which is well-suited for the efficient processing of large-scale graphs.

Keywords: Graph Partitioning, Large-Scale Graphs, Distributed Computing, GPU Acceleration.

Introduction:

Large-scale graphs represent a huge amount of data using nodes and edges. They have the ability to model complex relationships between objects and handle various sizes of structured data. The applications of large-scale graphs are found in social network analysis, recommendation systems, drug discovery, and knowledge graph reasoning. Efficient processing allows us to extract useful knowledge from these systems in a realistic time, and it will allow applications to handle millions or billions of nodes and edges, scale without exponential increases in execution time, avoid frequent system redesign, and result in less memory usage of the GPU and CPU. Processing a large-scale graph is very challenging and requires a significant amount of memory, time, and resources. To save time and resources, the large-scale graph is divided into subgraphs, where each subgraph contains a subset of nodes and edges. After that, the set of nodes is partitioned into mutually exclusive groups. Edges of the original graph that cross between the groups will produce edges in the partitioned graph. Large graphs cannot be represented on the GPU or even in the main memory as a whole. Poor processing results in Out-of-memory errors, unnecessary transfers between CPU and GPU, and wasted computational resource. For this purpose, SMEP is a new graph partitioning framework that aims to connect traditional partitioning methods with efficient runtime execution for large-scale graph processing. While most existing algorithms focus on optimizing structural metrics, SMEP takes an execution-oriented approach. It considers graph partitioning, memory estimation, execution needs, and thread grouping all at once during partition construction. This approach allows the resulting partitions to run through **the** thread grouping mechanism directly on GPU platforms, which helps cut down on repeated data transfers, synchronization delays, and memory waste.

Popular multilevel partitioning algorithms like METIS and KaHIP focus on reducing edge cuts and balancing partition sizes by coarsening and refining the graph. These methods create high-quality partitions but do not address GPU memory limits or runtime scheduling. Cluster-GCN, for example, speeds up graph neural network training by clustering the graph into mini-batches, but its main goal is to improve GNN training, not general-purpose partitioning or execution management. SMEP, on the other hand, combines partition creation with execution-aware scheduling. It creates execution packs that take GPU memory, resource use, and execution dependencies into account.

SMEP also uses a streaming partition construction method along with thread grouping. This lets large graphs be processed without needing the whole graph in **the** GPU memory at once. The design boosts scalability while keeping partition quality and making good use of resources. By bringing together graph partitioning and execution-aware scheduling in one framework, SMEP offers a practical solution for large-scale GPU-based graph processing that goes beyond what current partitioning frameworks can do. Overall, the central contribution of our work is as follows:

A graph partitioning algorithm is designed for large-scale graphs to minimize information loss and improve load imbalance.

A new thread grouping mechanism is proposed to enable parallel execution **on** heterogeneous systems.

Related Work:

Training and inference in Graph Neural Networks (GNNs) also require efficient processing of large-scale graphs, concentrating on the concept of parallel and distributed computation to boost performance. In addition, the need to incorporate pipelining and staleness management into GNN computation is mentioned [1]. The issue of excessive memory usage during graph partitioning, particularly on large graphs, as discussed in the paper, is one of the major hindrances to the effective scaling of algorithms. TeraPart is a multilevel graph partitioning technique that uses low memory footprints compared to other partitioning

algorithms, and does not compromise the quality or speed of the solution [2]. Omolayo et al. discuss why graph partitioning is a crucial concept to improve the scalability of distributed graph databases, load balancing, and query performance. The purpose of this paper is to systematically compare four partitioning algorithms (Random Vertex, Metis-based, Random Edge, and HDRF) between edge-cut and vertex-cut paradigms, based on their performance in different graph structures and workloads [3]. The large computational needs and the required memory in the GPUs limit GNNs to balance the efficiency of execution and prediction accuracy, especially when working with large-scale graphs. The LPS-GNN framework employs performance and efficiency enhancement strategies, graph partitioning, and subgraph augmentation techniques [4]. The challenges that dynamic graphs create to represent real-life entities and their ever-evolving relationships require an efficient processing capability to accommodate these changes and generate timely analysis output [5]. Expensive communication and low GPU utilization are major challenges in current systems used to train graph neural networks (GNNs) on large graphs. This paper suggests a framework named Distributed Sampling and Pipelining, which employs a custom data arrangement and proposes a collective sampling primitive to increase GPU usage and reduce communication expenses [6]. The underlying issue of assigning a graph into blocks that are of approximately equal weight with the least number of edge cuts is vital in parallel processing applications. The paper presents deep multilevel graph partitioning, which combines recursive bipartitioning and direct k-way partitioning to improve performance and quality when block numbers are large [7]. The difficulty of defining the graph partitioning algorithm as a multi-objective optimization task, which involves load balance and communication balance. The paper treats graph partitioning as a multi-objective constrained optimization problem with the emphasis on load balance and the communication overhead. In order to obtain the best partitioning result, an improved version of the NSGA-II algorithm is offered [8]. High-quality graph partitioning is important to achieve optimal performance. The current methods do not fully exploit the capabilities of modern GPUs, suggesting that further work on applying parallel algorithms to partition refinement is possible. Jet is a new parallel algorithm designed to refine partitioning on GPUs, with the aim of achieving higher quality than state-of-the-art algorithms [9]. The challenge of distributed training of GNNs on billion-scale graphs, focusing on minimizing cross-machine communication through effective partitioning algorithm [10]. Deep learning workload scheduling challenges related to GPU datacenters. The purpose of the document is to describe current deep learning workload scheduling approaches, highlighting their objectives, their benefits and the need for more efficient scheduling techniques [11]. Using multilevel refinement strategies, traditional graph partitioning algorithm such as METIS and KAHIP focus on producing high-quality partitions. METIS performs a lot of tremendous work and is still considered one of the most efficient algorithms. However, these methods are increasingly unstable for large-scale graphs due to their high memory requirements, global, and multiple refinement passes [12][13][14][15]. Cache contention in GPUs because of massive parallelism has been shown to cause performance degradation and energy waste. The paper presents PAVER, a Priority-Aware Vertex scheduler, which is theoretically graph-based on optimization of thread scheduling by analyzing the cache locality of thread blocks [16]. Y. Lu et al. discuss the difficulties of high performance with respect to the use of GPUs in graph processing, such as load imbalance and underutilization of hardware. GraphPEG: A graph processing engine that improves the performance of graph processing through a combination of automatic collection of edges with fine-grain distribution of work [17]. The complexity of working with large-scale graphs, especially at irregular access patterns in memory, creates performance decay in both CPUs and GPUs. FPGA processing engine that maximizes data transfers and a framework for using FPGA clusters, in an effort to increase their performance by optimally partitioning graphs [18]. Accomplishing high performance on fine-grained

parallel architectures that have traditionally not supported large-scale graph computations. Through tests and simulations of graph algorithms such as Breadth-First Search and PageRank, showing notable gains in performance over current systems [19]. In applications such as fraud detection and social network analysis, the NP-hard problem of temporal subgraph matching in large graphs is essential. In order to improve performance and scalability, this paper presents GTSM, a GPU-optimized system with a heterogeneous BFS-DFS execution model, memory-bound optimization, and a multi-edge-centric paradigm [20]. Large-scale graph processing systems face the difficulties of GPU memory over-subscription and the requirement for effective CPU-GPU cooperative processing. In order to overcome GPU memory constraints, this paper proposes CGgraph, an ultra-fast CPU-GPU graph processing system that balances CPU and GPU workloads and extracts a reusable subgraph [21]. Effective handling of massive streaming graphs that are larger than the GPU's memory capacity, resulting in significant data transfer overhead from the CPU to the GPU because of redundant graph accesses during continuous computation. Grapin is an out-of-memory GPU streaming graph processing system that uses a lightweight GPU hot subgraph management framework [22]. The challenges of efficiently processing dynamically changing graph data are highlighted by the growing demand for large-scale concurrent graph processing due to the growth of user scale and application data. CGCGraph is a solution designed to transfer unshared graph data to the CPU [23]. Ineffective state propagation of active vertices along the graph causes long convergence times in out-of-GPU-memory systems for iterative processing of large-scale graphs. LargeGraph is a system that improves parallelism paths and reduces data access costs by using a dependency-aware data-driven execution approach to increase the efficiency of state propagation [24]. In order to minimize redundant operations and maximize GPU utilization, this paper proposes a novel processing scheme that uses dynamic scheduling and operation reduction to process large-scale dynamic graphs efficiently, especially when dealing with repeated data transmission and processing during operations [25]. Graph processing faces challenges such as low communication ratios and poor data locality. Addressing these issues is crucial for improving performance in real-world applications that rely on efficient graph processing. ASGraph, a ReRAM-based graph processing accelerator based on dependency-sensitive [26]. The problem of subgraph enumeration is a very important issue in graph analytics that seeks to locate all instances of a query graph within a larger data graph. HUGE has an enhanced execution plan optimizer, a hybrid communication layer, and a two-stage execution mode. It uses a BFS/DFS-adaptive-scheduler that can manage memory consumption [27]. Efficient memory access for Out-of-memory graph traversal in GPUs (EMOGI) is a method for efficient graph traversal in GPUs that addresses the challenges of accessing large graphs stored in host memory, improving performance through optimized memory [28]. Graph compression has pointed to the possibility of direct processing of compressed graphs, although issues are still present about the memory usage and compression time. The introduction of Laconic solves these problems and offers a solution that minimizes memory usage [29]. The challenge of efficiently sampling graphs for representation learning algorithms is crucial for training deep neural networks (DNNs), because it can constitute a large portion of the overall training time and existing systems struggle to parallelize this process effectively. This paper introduces NextDoor, a system designed to perform graph sampling on GPUs using a novel approach called transit-parallelism, which enhances load balancing and edge caching [30].

Streaming Multilevel Execution Partitioning (SMEP):

SMEP is the new proposed graph partitioning algorithm, which is designed to bridge the gap between graph partitioning and efficient runtime execution in large-scale graph processing. The proposed algorithm introduces an integrated execution-oriented partitioning

strategy that simultaneously considers memory estimation, execution awareness, and partitioning rather than optimizing only the structural properties of the graph.

In a single-pass system, the algorithm sees each edge exactly once. It does not store the graph in a temporary buffer to look at it again later. This allows the system to process the graphs that are much larger than the available RAM. SMEP handles massive graphs in a single pass. In order to partition a graph, there is a need to keep track of where every node has been placed. For a graph with 500 million nodes, this mapping table can be massive. SMEP uses a compact data structure to ensure that the metadata does not become a bottleneck that crashes the system. Execution awareness is the most critical feature of SMEP. Traditional partitioners might create partitions that are perfect but 12GB in size. If the GPU only has 8GB of VRAM, that partition is useless. Treats memory as a hard constraint. If a partition reaches the GPU's memory limit, SMEP will stop adding data to it, even if it results in a slightly higher edge cut. This ensures that the graph runs without Out of Memory (OOM) errors. Low overhead, many partitioning tools (like METIS) perform an initial rough cut and then spend a long time refining the boundaries to improve it better. SMEP skip this. The goal is to get the graph ready for computation as quickly as possible. SMEP aims for good enough partitions as fast as possible. The overall farmwork of the proposed work is visually mentioned in Figure 1. SMEP operates in three conceptual stages:

Initialization: In lines (1-4), the algorithm starts initialization and does not process the entire graph at once. Breaks the list of edges into chunks(C). Instead of processing one edge at a time (which is too short-sighted), it looks at a small batch of edges. Tracks part_sizes and neighbor_stats (where a node's neighbors are currently located).

Streaming Multilevel Assignment: Lines (7-9), for the very first chunk of data, the algorithm has no history to go on. It uses a Seed Partition function to spread the nodes across the partitions P.

Execution-Aware: In lines (11-20), for every subsequent chunk, the algorithm performs a balancing act for every unassigned node (v). Neighbor Affinity calculates how many of node v's neighbors are already in each partition, and the goal is to minimize communication. Capacity penalty; here the execution-aware design comes in to play. The algorithm determines a penalty according to the completeness of a partition. Penalty is very high in case a partition is almost full.

Partition Metadata:

The partition metadata is important in partitioning the graph prior to the start of any computation. Its main role is to map out a small and effective map where the individual bits of the graph are located. Simply put, it identifies each node with a node partition ID (V) that is associated with a partition ID (P), establishing a distinct vertex-to-partition connection. This mapping is necessary as soon as downstream processing begins when the system needs to know fast the partition that has to provide information about a particular node. Instead of using complex or storage-intensive solutions, such as large databases, the solution is lightweight and intentional. This not only lessens the memory overhead but also makes lookup quicker during execution. This will result in a system that knows where to send or get data without unnecessary complexity or delays, making graph processing efficient and smooth.

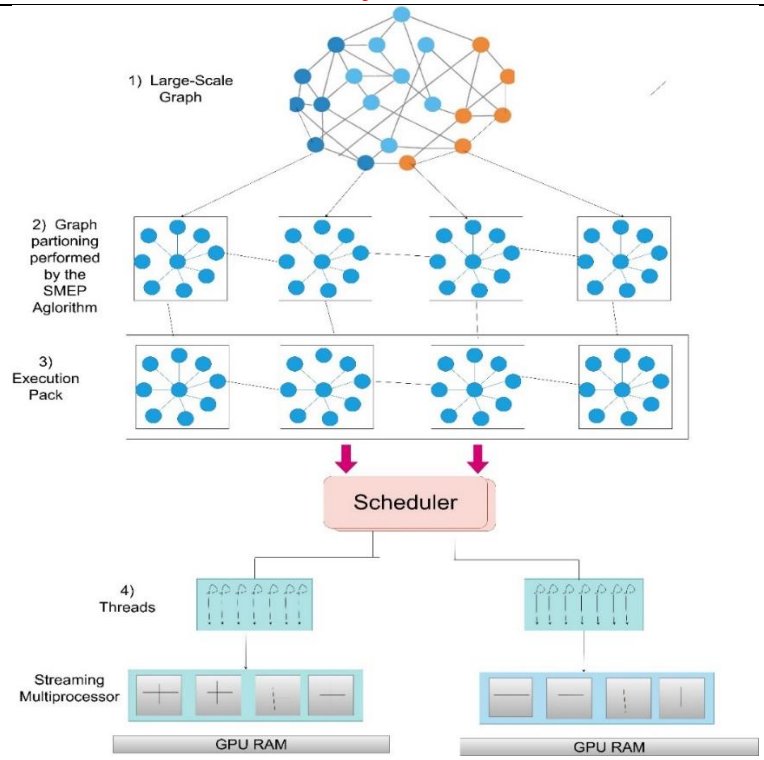

Figure 1. overview of the proposed SMEP-based methodology, from streaming graph ingestion through partitioning thread grouping, execution-pack formation, and scheduled GPU execution.

Algorithm 1: SMEP Algorithm

Input: G = (V, E), number of partitions k, batch size B, soft capacity C_{soft}
Output: Partition assignment array parts[v] for all v ∈ V
Initialize parts[v] ← ⊥ for all v ∈ V
Initialize part_size[p] ← 0 for all p ∈ {0, …, k−1}
Initialize neighbor_stats[v][p] ← 0 for all v ∈ V, p ∈ {0, …, k−1}
Partition edge list E into batches E_1, E_2, …, each of size B
for each batch E_i (in order) **do**
  $nodes_i$ ← unique vertices incident to E_i
  if i = 1 **then**
    parts[v] ← SeedPartition(E_i, k) for each v ∈ $nodes_i$
    Update part_size and neighbor_stats using E_i
  else
    for each v ∈ $nodes_i$ **do**
      if parts[v] = ⊥ **then**
        scores ← NeighborAffinity(v, E_i, parts)
        penalty[p] ← part_size[p] / C_{soft} for all p
        adjusted[p] ← scores[p] / (1 + penalty[p]) for all p
        p* ← argmax_p adjusted[p]
        parts[v] ← p*
        part_size[p*] ← part_size[p*] + 1
      end if
    end for
  end if
end for

Time and Space Complexity Analysis:

Let $|v|$ and $|E|$ denote the number of vertices and edges of the input graph G = (V, E), respectively; let P denote the number of partitions and let C denote the edge-chunk size, so that the number of chunks processed is $[|E|/C]$ and let d(v) denote the degree of vertex v.

Table 1. summarizes the operation and cost contribution by each line

Line	Operation	Cost
3	Initialize parts[v] = −1 for all v	$O(\|V\|)$
4	Initialize part_sizes[p] = 0	$O(P)$
5	Initialize neighbor_stats[v][p] = 0	$O(\|V\| \cdot P)$
6	Partition E into chunks of size C	$O(\|E\|)$
7-9	⌈|E|/C⌉ chunks; extract unique vertices per chunk	$O(\|E\|)$ total (hash set)
10	SeedPartition on first	$O(C)$
11	Update part_sizes / neighbor_stats after seeding	$O(C)$
13-21	For each unassigned vertex: compute NeighborAffinity (length-P score vector), apply penalty, take argmax over P	$O(P)$ per vertex

Time Complexity: By aggregating the per-line costs presented in Table 1 and applying the previously discussed amortization argument, the total running time of SMEP is determined as follows:

$$T(|V|, |E|, P) = O(|V|) + O(|V| \cdot P) + O(|E|) + O(|V| \cdot P)$$

$$T = O(|E| + |V|.P)$$

This bound is linear in the input size (|E|) plus a term linear in |V| scaled by P. Consequently, SMEP operates as an effectively linear-time algorithm.

Space complexity: Table 2 summarizes the space complexity of each algorithm component by defining them partially.

Table 2. Lists the auxiliary data structures maintained by SMEP and their memory footprint.

structure	size
parts[v]	$O(\|V\|)$
part_sizes[p]	$O(P)$
neighbor_stats[v][p]	$O(\|V\| \cdot P)$
Current chunk buffer Ei	$O(C)$ (streamed, not held in full)
Input graph	$O(\|V\| + \|E\|)$

Aggregating these terms, the overall time complexity of SMEP is:

$$S = O(|V|.P + |E|)$$

The dominant auxiliary term is $O(|V|.P)$, arising from the dense $|V| \times P$ neighbor stats; the streaming chunk buffer contributes only O(C) at any given time, since the full edge set need not be resident in memory.

Memory Estimation:

Beyond simple metrics including the number of nodes, memory estimation provides an algorithm an execution-focused perspective. It determines the actual memory footprint, or the number of bytes required to store an entire partition, compared to just counting the n umber of vertices. This comprises not just the nodes but also the edges connected to them, which may account for a significant portion of the overall memory usage. This estimation is directly related to the part_size value that will be utilized in SMEP algorithm. By constantly evaluating memory requirements as partitions are constructed, the system obtains a realistic view of resources utilization. The real-world advantages is that it has no limitations by the current memory constraints, which is typical problem associated with conventional partitioning techniques. In this way, it provides a smoother execution, fewer runtime errors, and more stable handling large-scale graph processing.

Cut Edge Tracking:

Whenever an edge's two ends are in neighboring partitions, it is commonly referred to a cut edge. As an example, when one node (A) is on a single GPU and another node (B) is on another. These edges are expensive, rather than enjoying the fast, local access to memory, and add latency. This part is constantly measuring the edge-cut, which provides a real-time measurement of the quality of the graph partitioning. A low ratio suggests that most of the connected nodes are kept together, leading to efficient computation. An increasing number of cut edges is an indication that an excessive number of connections are being divided between partitions. Once this occurs, the algorithm reacts by changing its strategy, and tries to retain adjacent nodes in the same partition, as long as the memory can accommodate it. By doing so, the state-of-the-art tracking does not only assess the performance, but also helps steer the partitioning process towards a more communication-efficient structure.

Thread Grouping:

The SMEP framework suggests a technique for organizing graph partitions into execution packs of threads before GNN training begins. Rather than distributing each graph partition across an independent execution thread, partitions that share a high level of connectivity and conform to the GPU's memory limitations are combined into one execution pack. The procedure involves selection of the largest partition as the starting point for an execution pack. Next, the remaining partitions are sorted according to their connectivity score, which is calculated as the ratio of the number of edges between partitions and their memory requirement estimate. The partition with the highest score is added to the pack sequentially until the total memory consumption of all the members exceeds the available GPU memory. When no further partition can be added, a new pack is initialized, and the procedure is repeated.

Scheduling Policy:

The scheduling policy is based on a memory-aware execution model. Execution packs are scheduled in parallel, and before running an execution pack, the scheduler predicts its memory requirements and checks whether there is enough memory on the GPU after reserving a safety margin. In case there is enough memory, the scheduler merges all partitions in the execution pack into one subgraph and executes the subgraph. Otherwise, the scheduler automatically resorts to running the partitions separately without failing due to lack of memory, thereby keeping execution seamless. There is another stage of adaptive elastic scheduling, where the initial execution packs are further optimized by merging small execution packs or splitting large ones depending on the available GPU memory.

Synchronization Strategy:

The thread-grouping strategy relies on lightweight synchronization to handle data movement and computations. When GPU prefetching is enabled, the execution pack is transferred to device memory using a CUDA stream. At the same time, the default execution stream processes the previous execution pack. The synchronization points on the stream, makes sure all memory copies finish before the kernel runs. After training on the current execution pack ends, the GPU caches are cleared, memory is synchronized, and the next execution pack is scheduled.

Execution Pack:

Graph processing has been performed [31][32][33][34][35][36]. After the GP and thread grouping, there is a need to process these partition schemes in parallel on GPUs. Here, we need to devise a thread-grouping mechanism and an execution pack. An execution pack dynamically adjusts the workload distribution the workload distribution at runtime based on available resources and execution imbalance. In graph partitioning, workloads are often irregular (because graphs are sparse, dense, and skewed). Some partitions end up being heavier (more edges, higher-degree nodes) than others. EP ensures work stealing, migration, and

dynamic scheduling are possible, so overloaded processors hand off tasks to underloaded ones. Lines 1-7. It starts by sorting partitions by size (M(p)), from largest to smallest.

Algorithm 2: Thread Grouping

1: **Input:** Partition set $\mathcal{P} = \{0,\ldots,k-1\}$, memory function M(p), pairwise weight function W(p,q), pack capacity C
2: **Output:** Set of execution packs $\Pi = \{pack_1, \ldots, pack_K\}$
3: Sort $\mathcal{P}$ by decreasing M(p)
4: Unassigned $\leftarrow \mathcal{P}$
5: $\Pi \leftarrow \emptyset$
6: **while** unassigned $\neq \emptyset$ **do**
7: seed $\leftarrow \text{argmax}_{p \in Unassigned}$ M(p)
8: pack $\leftarrow$ {seed}; mem $\leftarrow$ M(seed)
9: Remove seed from Unassigned
10: **while** true **do**
11: $q^* \leftarrow \text{argmax}_{q \in Unassigned}$ W(q,pack)/M(q), where $W(q,pack) := \Sigma_{p \in pack}$ W(q,p)
12: **if** Unassigned = $\emptyset$ **then break**
13: **if** mem + M(q*) $\leq$ C **then**
14: pack $\leftarrow$ pack $\cup$ {q*}; mem $\leftarrow$ mem + M(q*)
15: Remove q* from Unassigned
16: **else**
17: **break**
18: **end if**
19: **end while**
20: $\Pi \leftarrow \Pi \cup$ {pack}
21: **end while**
22: **return** Π

The seed picks the largest remaining partition to start a new pack. In line (10), the algorithm selects the partition (q) to add to the current pack by maximizing the following ratio:

$$\frac{w\ (q, \text{pack})}{M(q)}$$

W (q, pack): The strength of the connection (edge weight) between partition q and the partitions already in the pack. M(q) is the memory size of partition q. Lines (11–15), before adding partition q to the pack, it checks:

$$\textbf{mem + M (q)} \leq \textbf{C}$$

Where C is the GPU's total memory capacity. If the pack is already too full to fit the partition q, it closes that pack and starts a new one. This ensures that no pack will ever trigger an out-of-memory error when loaded onto a GPU.

Time complexity: Thread Grouping runs in $O(n^2)$ time when the scores W (q, pack) are maintained incrementally or $O(n^3)$ if recomputed from scratch each trial, where n is the number of partitions. This bound comes from two independent sources that happen to coincide in order: the $O(n^2)$ cost of shifting elements during array-based removal, and the $O(n^2)$ (cached) or $O(n^3)$ cost of scanning Unassigned to select the best score at each step. The O(n log n) sort and the O(n) seed-selection cost are dominated by these terms.

Space complexity: The algorithm uses $O(n^2)$ space mainly because of the dense pairwise weight matrix W(p,q). If weights are stored sparsely, space drops to O (n + m), where m is the number of partition pairs with nonzero weights. All other structures, such as P, M(p), Unassigned, pack, and execution packs, need only O(n) space.

Work Stealing:

Each processor maintains its own partition queue. When a processor becomes idle, it looks for other busy processors and attempts to take a partition from one of their queues, as shown in Figure 2, where the processor's queue is filled with partitions p1, p2, p3, p4, p5, and p6, which represent busy processors, while another processor represents the idle queue, which is looking to steal partitions from the busy queues to steal them and fill the idle queue. This approach is highly scalable because it avoids a single, centralized scheduler. It is often used in systems with a high degree of parallelism and is especially effective for irregular workloads where task run times are unpredictable.

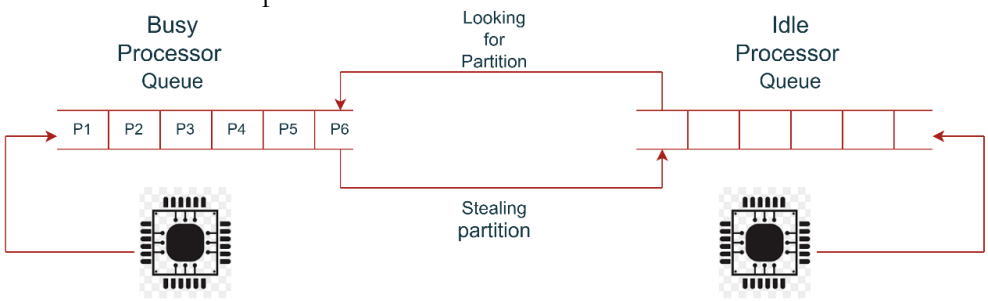

Figure 2. Work-Stealing mechanism: idle preprocessors pull partition from the queues of busy processors to keep the GPU utilization high under irregular workloads

Work Migration:

In graph processing, a processor is considered overloaded when it has too many active nodes to process, experiences high communication due to many edges crossing to other partitions, has long message queues waiting to be processed, or faces GPU memory pressure. Once overload is detected, the system selects a suitable graph partition to migrate, such as a subgraph with fewer connections to other partitions or a cluster with minimal cross-partition dependencies. The graph partition is then distributed to various processors, either across a network in a distributed system or in a multi-GPU system.

Dynamic Scheduling:

Dynamic scheduling operates differently. It does not make scheduling decisions on a fixed basis, but in response to the current state of the system, the length of processor queues, the activity of the devices, the pressure of the memory, and even the delays in the communications. Two mechanisms, namely work stealing and migration, make this possible. These techniques transform scheduling from a static to a dynamic process. This results in increased resource utilization, decreased idle time, and a responsive system.

Requirement of Execution Pack:

Mapping: Each subgraph must be mapped to the CPU cores or GPU's memory blocks, or even many GPUs after partitioning. These mappings are arranged into units of work by the execution pack for effective dispatch.

Dependency: When a graph is divided into several partitions, dependency management becomes essential. Some partitions rely on data generated by others, especially when cross-partition edges are present. In these situations, computations proceed at random and the execution pack carefully organizes the sequence of computation. In order to ensure that any necessary data is available before a partition starts working, dependencies are handled first. As a result, there are fewer synchronization problems and less idle waiting, resulting in a controlled execution flow.

Load Balancing: Some partitions are very light, while others contain extensive features and dense connections. By dynamically altering the arrangement and processing of partitions, the execution pack corrects this imbalance. Lighter partitions are combined to preserve efficiency, while heavier ones are divided or scheduled.

Communication Minimization: Instead of scheduling partitions randomly, the execution pack carefully groups them in a way that minimizes the need for frequent data exchange across GPUs. By keeping computations within memory limits and avoiding excessive data movement, the execution pack enables efficient execution, better resource utilization.

Parallelism Control: Due to data dependencies or GPU limitations, not all partitions can run concurrently. To maximize concurrency without compromising correctness, the execution pack divides partitions into execution rounds.

Execution Round:

An execution round is the scheduling cycle in which execution packs are processed in parallel on available hardware. It is required to respect memory limits, synchronization dependencies, and load balancing. Not all packs can be run simultaneously; rounds allow controlled scheduling across available devices.

Need for Execution Round**:** Not every execution pack can be executed at the same time, rounds can be scheduled on available devices.

Parallelism Control: Each round executes as many packs as hardware allows because rounds maximize concurrency while respecting synchronization barriers.

Load Balancing: Rounds help to spread irregular workload. If one pack is very heavy, the scheduler can pair it with lighter packs in that round.

A Single GPU executes packs sequentially, round by round. Each round is equal to one pack. In multi-GPUs, dispatch as many EPs as GPUs. Similarly, in a distributed system, the packs are grouped per node to minimize cross-node communication, and synchronization occurs at the end of each round before moving to the next.

Scheduler:

After graph partitioning, execution pack, and execution round, the scheduler decides when, where, and how each pack is executed across a heterogeneous system. A visual representation of the scheduler is shown in Figure 3.

Working of the Scheduler**:**

Input to scheduler: The scheduler receives the execution pack, hardware resource information (e.g., number of GPUs, memory per GPU), and dependencies (boundary node communication and inter-pack communication). The scheduler assigns each EP to a GPU device and ensures that each pack fits the memory constraints. The core responsibility of the scheduler is given below:

Round Formation: Groups of packs are organized into execution rounds based on available devices. If 4 GPUs exist and 12 packs → 3 rounds (4 packs per round).

Communication Scheduling: Minimizes cross-GPU data exchange and schedules communication overlap with computation (while GPU0 computes, GPU1 transfers boundary data).

Dynamic Load Balancing: Some partitions are heavier; a scheduler can split heavy packs or pair them with lighter ones.

Fault Tolerance: When a GPU or node fails or starts to perform poorly. The scheduler reacts by redistributing the affected execution pack to other available devices.

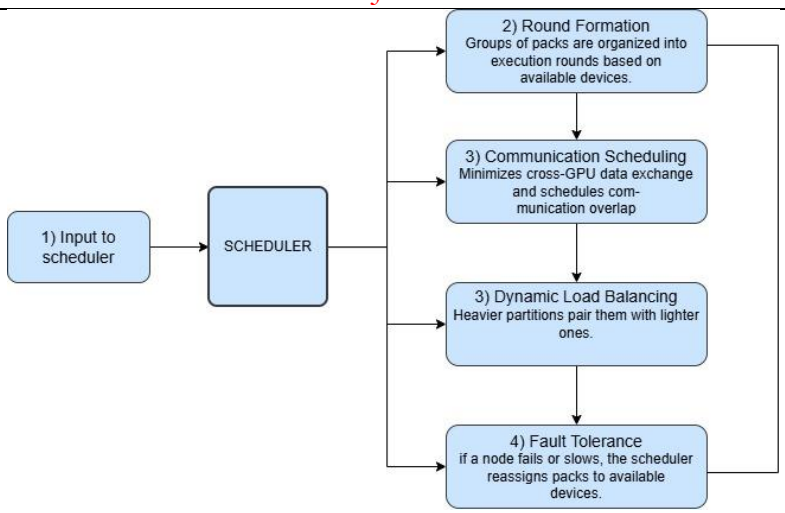

Figure 3. Responsibilities of the scheduler in coordinating round formation, communication scheduling, dynamic load balancing, and fault tolerance across GPU device.

Result and Discussion:

The practical implementation of the proposed work is discussed in this section, which turns the methodology into an executable system that can effectively train large-scale graph datasets on GPU hardware. Table 3 provides a detailed description of all the datasets used in the work, which are widely used benchmark datasets that represent large-scale real-world graphs with varying structural properties.

Table 3. Dataset Characteristics

Dataset	Nodes	Edges	Features	Classes
OGBN-PR	2.45M	61.8M	100	47
Reddit	232,965	11.6M	602	41
LiveJournal	4.85M	68.9M	None	None

Dataset Preprocessing:

For the purpose of assessing the framework, three openly accessible benchmark datasets, namely Reddit, OGBN-Products, and LiveJournal, were chosen since they represent large-scale graphs with varied structural properties. Prior to the partitioning, each dataset was first converted to PyG format to enable compatibility between the framework and the data. Since some datasets did not include full information regarding the node features and labels, degree-based features and dummy labels were generated to ensure a consistent dataset format throughout the experiments. Similarly, where predefined train/validation/test masks were missing, the nodes were randomly partitioned into training (60%), validation (20%), and test (20%) partitions.

After preprocessing, the graph datasets were partitioned using the proposed Streaming Memory-aware Execution Partitioning (SMEP) algorithm. The partitions created are self-contained while maintaining the necessary structural connections to enable effective GNN learning. The partitions are then further utilized within the execution framework to assess the performance of the proposed solution in terms of execution times, memory usage, and model accuracy.

Environmental Details:

The local evaluation of the new SMEP algorithm, its comparison with METIS, and the execution times for SMEP thread group formation were carried out on 8 machines in the local environment. As METIS is CPU-centric, it cannot perform partitioning on the GPU due to the GPU's limited memory. SMEP uses a multi-core system to explore the complex structure of the large graphs, enabling efficient graph partitioning into smaller subgraphs. For parallel execution of packs, the framework uses the GPU's hardware. This is crucial as

fundamental libraries such as PyTorch Geometric and DGL are tightly linked to the CUDA (Compute Unified Device Architecture) environment. Additionally, the thread grouping policy (hardware optimization) demands fine-grained control over threads, which is achieved through the ZOTAC GAMING GeForce GTX1660 SUPER GPU. The experiments are conducted with Python 3.12, PyTorch 2.3.0, PyTorch Geometric 2.x, and CUDA 12.0.

GraphSAGE:

GraphSAGE is preferred because it is capable of doing inductive learning: therefore, new node embeddings can be produced without training the whole network. It leverages a sampling and aggregation mechanism to learn generalized functions that generate embeddings for previously unseen nodes or subgraphs. Through fixed-size neighborhood sampling, the memory usage of each mini-batch is consistent. It is possible for frameworks to perform mini-batch training on large graphs that have billions of nodes, all while remaining within available GPU VRAM. GraphSAGE brought about mini-batch training capabilities for graphs. GraphSAGE design patterns are implemented in frameworks such as PyTorch Geometric (PyG) and DGL natively. In real-world systems, there is a creation of new nodes, through GraphSAGE, frameworks can infer embeddings of new nodes by passing them through the network once. In contrast to the GCN model, which generally needs graph propagation on the entire graph, GraphSage uses the technique of neighborhood sampling, which fits perfectly well with the suggested partitioning strategy. In comparison with the GAT model, GraphSAGE has considerably lower computational complexity since it does not use attention mechanism for all edges of the graph. Despite having better expressive power than GraphSAGE, the GIN model is computational expensive.

Comparison with Reddit:

Based on the experimental results for the Reddit dataset, here is a detailed analysis of the two algorithms mentioned in Table 4. At 4 partitions, SMEP completes the task in 56s, whereas METIS takes 450s. This is a 10.6x speedup. METIS time fluctuates significantly while SMEP remains incredibly stable. Throughout all tests, SMEP maintains a partitioning time between 56s and 68s. As the number of partitions increases from 4 to 128, SMEP's execution time decreases by 12s (21%). METIS shows an unusual trend where it is slowest at the lowest partition count.

Table 4. SMEP vs METIS Partitioning time for the Reddit

Partitions	Metis(s)	SMEP(s)
4	450	**56**
8	487	**57**
16	490	**59**
32	491	**61**
64	500	**58**
128	501	**68**

This shows that METIS struggles with the initial global structure analysis of the Reddit graph. However, the SMEP algorithm handles the initial data ingestion much more efficiently. SMEP achieves its 10x speed advantage without requiring any additional memory compared to METIS, proving that it is a more computationally efficient algorithm for large-scale graphs.

Without thread grouping, METIS shows significantly higher execution times across all partition sizes. With 4 partitions, METIS takes 69.75 seconds, whereas SMEP completes in only 42.43 seconds. Even when the number of partitions is increased to 128, METIS still requires 240.38s, whereas SMEP takes just 105.06s, as shown in Table 5. This shows that SMEP consistently outperforms METIS in execution time. SMEP is an execution-aware design that reduces communication overhead and improves execution efficiency. is shown in Figure 4. With thread grouping, both methods show performance improvements. SMEP

maintains stable, lower execution times across all partitions (42.43–105.06s), whereas METIS still suffers from higher variability and higher overall cost (69.75-240.38s) without thread grouping.

Table 5. Total Execution time for the Reddit

Partitions	Metis(s)	SMEP(s)
4	69.75	**42.43**
8	78.37	**31.62**
16	90.25	**51.76**
32	109.39	**73.64**
64	153.39	**89.39**
128	240.38	**105.06**

This indicates that SMEP not only benefits from thread grouping but also complements it effectively by providing balanced workloads, thereby reducing thread divergence.

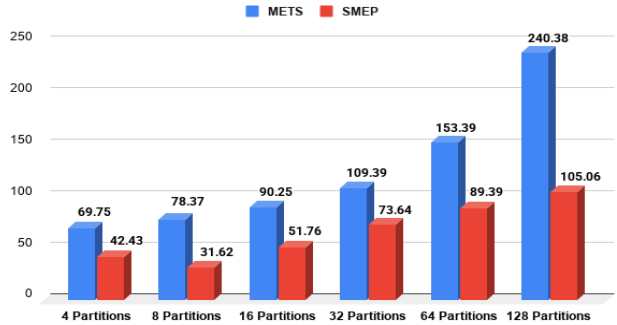

Figure 4. Total end-to-end execution time vs partition count for SMEP and METIS on the Reddit dataset.

Comparison with LiveJournal:

The comparison data for the LiveJournal dataset is a breakdown of the SMEP algorithm's performance against METIS, and these results are presented in Table 6. SMEP maintains a massive speed advantage, though the gap closes slightly as the number of partitions reaches the extremely high end. At 4 partitions, SMEP is nearly 10x faster than METIS (369 vs 418s).

For the configuration between 4 and 64 partitions, SMEP remains very fast (59 to 91s), while

METIS stays relatively stagnant around the 418 marks. Similarly, at 128 partitions, SMEP remains at XX s, while METIS jumps to 418 s, which shows that METIS is still struggling with the global structure of the LiveJournal graph.

Table 6. SMEP vs METIS Algorithm in LiveJournal

Partitions	Metis(s)	SMEP(s)
4	369	**59**
8	383	**64**
16	398	**70**
32	419	**75**
64	412	**81**
128	418	**91**

The execution time comparison for the LiveJournal dataset is shown in Table 7, which compares how well METIS and the SMEP algorithm perform in two different configurations: METIS shows significantly longer execution times without thread grouping for all partition sizes (71.12-54.06). This is due to METIS's lack of execution awareness, which ignores runtime communication overhead and GPU memory limitations. Consequently, uneven workloads and

frequent cross-partition communication result in longer execution times. SMEP, on the other hand, exhibits significantly reduced execution times even in the absence of thread grouping, thereby minimizing redundant data movement and better aligning partitions with hardware limits. The visual comparison of the results is shown in Figure 5.

Table 7. Total Execution time for the LiveJournal

Partitions	METIS(s)	SMEP
4	71.12	**54.06**
8	74.67	**41.95**
16	77.08	**50.78**
32	84.55	**64.57**
64	96.66	**72.93**
128	126.95	**84.77**

The similarity is further reinforced by the effect of thread grouping. TG enables SMEP, which displays shorter execution times for all partition sizes (54.06-84.77). While METIS + TG still has greater execution overhead and variability, the SMEP + TG configuration maintains comparatively constant and consistently lower execution speeds across all partitions. This suggests that because of its hierarchical partitioning and lower edge-cut overhead, SMEP is not only better suited to take advantage of GPU parallelism but is also better suited for thread grouping and efficient large-scale graph partitioning.

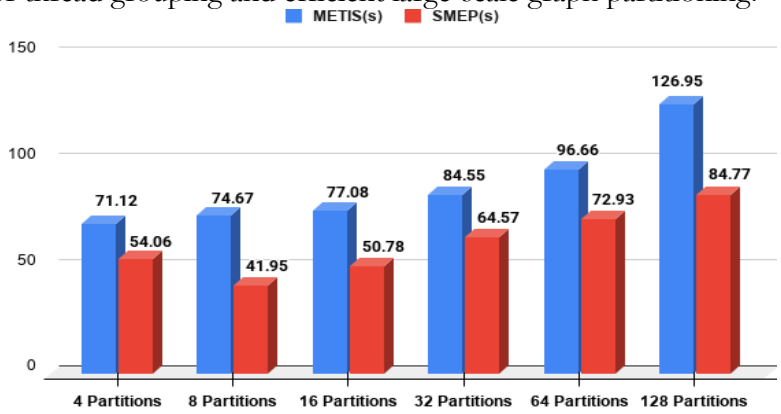

Figure 5. Total end-to-end execution time Vs partition count for SMEP and METIS on the LiveJournal dataset.

Comparison with OGBN-PR:

The comparison between the METIS and SMEP algorithms on the OGBN-Products dataset is presented in Table 8. At 4 partitions, SMEP takes only 67s, compared to 510s for METIS; this represents a speedup of approximately 11.7x. METIS incurs a high initial computational cost to analyze the global graph structure before it can start partitioning, whereas SMEP is much more immediate. METIS's performance leveled out after 8 partitions, staying consistently around the 586 marks. In contrast, SMEP's time increases as the number of partitions grows (from 67s at 4 partitions to 106s at 128 partitions). Even at the highest partition count (128), where SMEP is at its slowest (106), it is still 3.6x faster than METIS (586).

Table 8. SMEP vs METIS Partitioning Time in OGBN-PR

Partitions	Metis(s)	SMEP(s)
4	510	**67**
8	577	**76**
16	580	**81**
32	583	**90**
64	588	**97**
128	586	**106**

Table 9 presents a comparison of execution times for the OGBN-PR dataset, evaluating METIS and SMEP in both the thread grouping (TG) and without-thread-grouping (W.TG) configurations. METIS shows substantially longer execution times under the no-thread-grouping arrangement for all partition sizes. For example, METIS takes 102.94 seconds with 4 partitions, while SMEP completes the same work in just 39.87 seconds, demonstrating a significant performance difference. Due to high communication overhead and a lack of execution awareness, METIS exhibits unstable, frequently increasing execution times (215.98 s at 128 partitions) as the number of partitions increases.

Table 9. Total Execution time for the OGBN-PR

Partitions	METIS(s)	SMEP
4	102.94	**39.87**
8	99.46	**34.91**
16	100.76	**49.87**
32	108.97	**62.81**
64	131.04	**74.95**
128	175.37	**98.67**

As demonstrated in Figure 6, SMEP maintains relatively shorter, more controlled execution times (98.67s at 128 partitions), indicating its capacity to handle memory more effectively and minimize cross-partition communication. Both approaches achieve shorter execution times when thread grouping is used due to better workload balance and reduced thread divergence. For SMEP, the improvement has a far greater impact. While METIS + TG still has higher overall execution times (102.94s at 4 partitions and 175.37s at 128 partitions), the SMEP + TG configuration consistently delivers lower total execution times across all partitions, including 98.67s at 128 partitions.

When Thread Grouping (TG) is applied, both methods benefit from reduced execution time due to improved workload balance and reduced thread divergence. However, the improvement is significantly more impactful for SMEP. The SMEP + TG configuration achieves consistently lower total execution times across all partition sizes—for example, 39.87s at 4 partitions and 98.67s at 128 partitions —while METIS + TG still suffers from higher overall costs (102.94s at 4 partitions and 175.37s at 128 partitions). Scalability-wise, METIS behaves poorly as partitions grow, with execution time increasing quickly, especially in the absence of TG.

The overall execution time is still much longer, even with TG. SMEP, on the other hand, offers more consistent and scalable performance, especially when associated with TG, where execution time progressively increases while remaining within a reasonable range. Even on local hardware, thread grouping stabilizes the execution time that stays almost constant as the number of partitions rises. The proposed approach confirms that TG reduces communication overhead for both local and cloud systems.

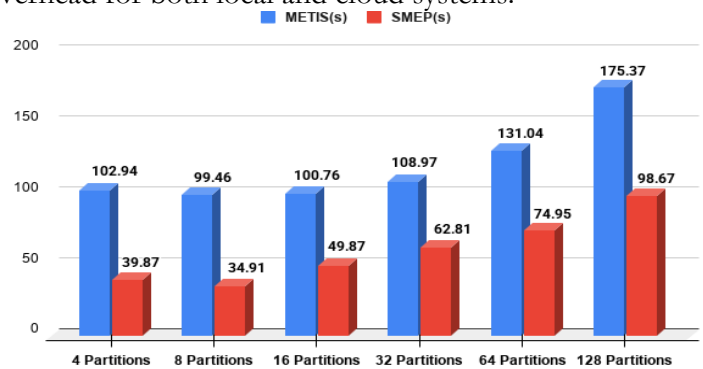

Figure 6. Total end-to-end execution time vs partitions count for SMEP and METIS on the OGBN-products dataset.

Cut-Edge Analysis:

Figure 7 compares SMEP and METIS in terms of cut-edge ratio across three datasets. On Reddit, SMEP achieves a cut-edge ratio of approximately 10%, whereas METIS records 18%. For OGBN-Products, SMEP attains 13% and METIS 24%. On LiveJournal, SMEP reaches 17%, compared to 28% for METIS. Consequently, SMEP reduces the cut-edge ratio by 44.4% on Reddit, 45.8% on OGBN-Products, and 39.3% on LiveJournal relative to METIS (see Table 10). Given that cut edges influence cross-partition communication, these reductions indicate that SMEP results in less information loss and reduced communication overhead.

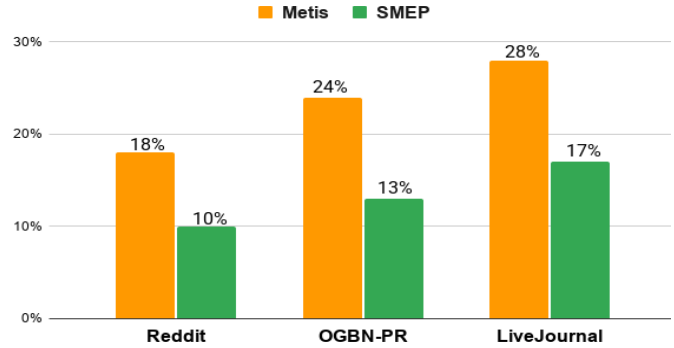

Figure 7. Cut-edge ratio produces by SMEP and METIS on the Reddit, OGBN-Products and LiveJournal datasets.

Comparison with Frameworks:

Table 10 demonstrates that current graph partitioning frameworks vary considerably in their support for GPU-oriented execution. METIS and KaHIP emphasize minimizing edge cuts and balancing partition sizes through multilevel partitioning, yet they do not account for GPU memory constraints, streaming partitioning, runtime scheduling, or memory-aware execution. Cluster-GCN employs graph clustering to enhance Graph Neural Network (GNN) training efficiency and offers partial GPU and memory awareness by generating mini-batches suitable for GPU memory. However, it does not facilitate streaming graph processing or runtime scheduling. DSP advances traditional partitioning by integrating dynamic streaming and execution scheduling with GPU memory restrictions and execution scheduling with GPU memory restrictions, although its streaming functionality is limited by batch-oriented operations in certain phases.

Table 10. Comparison with other frameworks

Framework	GPU-Aware	Streaming	Runtime Scheduling	Memory-aware
METIS	No	No	No	No
KaHIP	No	No	No	No
Cluster-GCN	Partial	No	No	Partial
DSP	Yes	Partial	Yes	Yes
SMEP	**Yes**	**Yes**	**Yes**	**Yes**

The SMEP framework, by contrast, comprehensively incorporates GPU-aware partitioning, streaming partition construction, runtime scheduling, and memory-aware optimization inside a unified execution-oriented system. SMEP produces execution-ready partitions and execution packs that account for GPU memory capacity and runtime resource usage, thereby decreasing data movement, synchronization load, and memory consumption while increasing scalability for large-scale graph processing. These combined features distinguish SMEP from present frameworks and constitute its principal contribution to efficient GPU-based graph analytics.

Resource Utilization:

The graph partitioning strategy initially runs on the CPU, requiring 8 to 9 GB of memory. After partitioning, thread grouping merges the partitions, and the execution pack is

processed on the GPU. For the Reddit dataset, this process uses 7.5 GB of GPU memory with 62% GPU utilization. The same approach is applied to the LiveJournal dataset, using 7.7 GB of GPU memory and 64% utilization. For the OGBN-Products dataset, 8.5 GB of GPU memory is used, and GPU utilization reaches 70% to manage the large-scale graph.

Limitations:

Despite major improvements in partitioning and execution efficiency in the proposed Streaming Multilevel Execution Partitioning (SMEP) framework, it still has several crucial limitations, including the following. First, the SMEP framework currently assumes a static graph structure and needs further extension to handle dynamic graphs with intensive modifications. Second, the scheduler of the execution pack is based on NVIDIA GPU architectures and CUDA streams and needs further modification to work with AMD GPUs, TPUs, or CPUs. Third, the proposed framework has been tested only on homogeneous graphs, and the partitioning and memory estimation algorithms should be further extended to keep the semantic information of heterogeneous graphs. Although the proposed streaming partitioning approach reduces execution time, its optimization objective prioritizes execution efficiency and memory-aware scheduling rather than achieving optimal edge cuts with advanced multi-level graph partitioners. Consequently, a trade-off exists between partitioning speed and minimizing communication, particularly for graphs with highly irregular topology. Despite these limitations, the proposed approach provides an efficient and scalable foundation for large-scale graph processing. It also establishes a platform for future extensions into dynamic, heterogeneous, and distributed graph learning environments.

Conclusion and Future Work:

In this paper, a novel Streaming Multilevel Execution Partitioning (SMEP) framework has been introduced to improve the efficiency of large graph processing through memory-aware graph partitioning and execution scheduling. While existing approaches to graph partitioning have mainly concentrated on achieving good partition quality, SMEP considers graph streaming partitioning along with thread grouping, execution-pack creation, and adaptive GPU memory scheduling. Through joint optimization of partitioning and execution scheduling, the proposed framework has been shown to minimize computational overhead and optimize the use of GPU memory resources. The experimental evaluation of the proposed method using three large benchmark datasets—Reddit, OGBN-Products, and LiveJournal—has proved its efficacy. The results showed lower execution time irrespective of the chosen partition size. Using the execution-pack technique, GPU memory requirements were minimized by executing strongly connected partitions together, thus minimizing redundant computation and increasing throughput. Moreover, the memory-aware scheduling scheme-maintained workload balance regardless of partition size while reducing inter-partition communication caused by edge cuts. This demonstrates that SMEP is an effective and scalable framework to process large graphs. However, although the framework achieves significant improvements, there are still avenues for further research. To make the SMEP framework more scalable and useful, it should be extended to work with multi-node distributed clusters. This can be done by improving the way partitions are allocated to reduce communication between nodes. At the same time, adding adaptive partitioning strategies will help process real-time streaming data and automatically rebalance changing graph structures with little migration overhead. The improved framework should also work smoothly with modern graph processing libraries such as PyTorch Geometric, DGL, and GraphBolt. This will help optimize Graph Neural Network (GNN) training by using better neighbor sampling, mini-batching, and memory scheduling.

Acknowledgement: The University of Peshawar contributed technical assistance, supervision, and access to computational resources that enabled this research, for which the

author is grateful. The dataset provider that provided the basis for this work is also acknowledged.

Author's Contribution: The corresponding author should explain the contribution of each co-author completely.

Conflict of Interest: Regarding the publication of this research study in the IJIST, the authors state that they have no conflicts of interest.

References:

[1] D. Salwasser, D. Seemaier, L. Gottesbüren, and P. Sanders, "Tera-Scale Multilevel Graph Partitioning," Oct. 2024, Accessed: Jul. 27, 2026. [Online]. Available: https://arxiv.org/pdf/2410.19119

[2] O. Omolayo, O. Oloruntoba, S. Adepoju, and K. Audu, "Comparative Analysis of Graph Partitioning Strategies for Enhancing Scalability and Performance in Distributed Graph Databases," Jun. 2025, doi: 10.21203/RS.3.RS-6900355/V1.

[3] X. Cheng *et al.*, "LPS-GNN : Deploying Graph Neural Networks on Graphs with 100-Billion Edges," *ACM Trans. Knowl. Discov. Data*, vol. 20, no. 5, Jul. 2025, doi: 10.1145/3801100.

[4] H. Gao, X. Liao, Z. Shao, K. Li, J. Chen, and H. Jin, "A survey on dynamic graph processing on GPUs: concepts, terminologies and systems," *Front. Comput. Sci. 2024 184*, vol. 18, no. 4, pp. 184106-, Dec. 2023, doi: 10.1007/S11704-023-2656-1.

[5] Z. Cai *et al.*, "DSP: Efficient GNN Training with Multiple GPUs," *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program. PPOPP*, vol. 1, pp. 392–404, Feb. 2023, doi: 10.1145/3572848.3577528;PAGE:STRING:ARTICLE/CHAPTER.

[6] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier, "Deep Multilevel Graph Partitioning," *Leibniz Int. Proc. Informatics, LIPIcs*, vol. 204, May 2021, doi: 10.4230/LIPIcs.ESA.2021.48.

[7] M. S. Gilbert, K. Madduri, E. G. Boman, and S. Rajamanickam, "Jet: Multilevel Graph Partitioning on Graphics Processing Units," Apr. 2023, Accessed: Jul. 27, 2026. [Online]. Available: https://arxiv.org/pdf/2304.13194

[8] R. Waleffe, D. Sarda, J. Mohoney, E.-V. Vlatakis-Gkaragkounis, T. Rekatsinas, and S. Venkataraman, "Armada: Memory-Efficient Distributed Training of Large-Scale Graph Neural Networks," Feb. 2025, Accessed: Jul. 27, 2026. [Online]. Available: https://arxiv.org/pdf/2502.17846

[9] H. Cui, F. Cao, and R. Liu, "A multi-objective partitioning algorithm for large-scale graph based on NSGA-II," *Expert Syst. Appl.*, vol. 263, p. 125756, Mar. 2025, doi: 10.1016/J.ESWA.2024.125756.

[10] Z. Ye *et al.*, "Deep Learning Workload Scheduling in GPU Datacenters: A Survey," *ACM Comput. Surv.*, vol. 56, no. 6, p. 146, Jun. 2024, doi: 10.1145/3638757;Journal:Csur;Wgroup:String:Acm.

[11] "(PDF) Parmetis: Parallel graph partitioning and sparse matrix ordering library." Accessed: Jul. 27, 2026. [Online]. Available: https://www.researchgate.net/publication/238705993_Parmetis_Parallel_graph_partitioning_and_sparse_matrix_ordering_library

[12] A. Gupta, "Fast and effective algorithms for graph partitioning and sparse-matrix ordering," *IBM J. Res. Dev.*, vol. 41, no. 1–2, pp. 171–183, 1997, doi: 10.1147/RD.411.0171.

[13] A. Pirova, I. Meyerov, E. Kozinov, and S. Lebedev, "PMORSy: parallel sparse matrix ordering software for fill-in minimization†," *Optim. Methods Softw.*, vol. 32, no. 2, pp. 274–289, Mar. 2017, doi: 10.1080/10556788.2016.1193177;REQUESTEDJOURNAL:Journal:Opms;Wgroup:String:Acm.

[14] L. Meng *et al.*, "A Survey of Distributed Graph Algorithms on Massive Graphs," *ACM Comput. Surv.*, vol. 57, no. 2, p. 39, Oct. 2024, doi: 10.1145/3694966;Requestedjournal: Csur;Taxonomy:Taxonomy:Acm-Pubtype;Pagegroup:String:Publication.

[15] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "PAVER: Locality Graph-Based Thread Block Scheduling for GPUs," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, Jun. 2021, doi: 10.1145/3451164;Page:String:Article/Chapter.

[16] Y. Lü *et al.*, "GraphPEG: Accelerating Graph Processing on GPUs," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, Jun. 2021, doi: 10.1145/3450440;Requestedjournal: Taco;Serialtopic:Topic:Acm-Pubtype.

[17] A. Sahebi, M. Barbone, M. Procaccini, W. Luk, G. Gaydadjiev, and R. Giorgi, "Distributed large-scale graph processing on FPGAs," *J. Big Data 2023 101*, vol. 10, no. 1, pp. 95-, Jun. 2023, doi: 10.1186/S40537-023-00756-X.

[18] Y. Wang, C. Colley, B. Wheatman, J. Su, D. F. Gleich, and A. A. Chien, "How Fast Can Graph Computations Go on Fine-grained Parallel Architectures," Jul. 2025, Accessed: Jul. 27, 2026. [Online]. Available: https://arxiv.org/pdf/2507.00949

[19] J. He, M. Jia, Y. Chen, Z. Liu, and D. Li, "GTSM: A multi-edge-centric temporal subgraph matching framework on GPUs," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 4, Dec. 2025, doi: 10.1145/3771286;Website: Dl-Site;Taxonomy:Taxonomy:Acm-Pubtype;Pagegroup:String:Publication.

[20] P. Cui, H. Liu, B. Tang, and Y. Yuan, "CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1405–1417, Feb. 2024, doi: 10.14778/3648160.3648179;Wgroup:String:Acm.

[21] Q. Wang *et al.*, "Efficient Graph Data Access for Out-of-Memory GPU Streaming Graph Processing," *Proc. VLDB Endow.*, vol. 18, no. 11, pp. 3854–3867, Jul. 2025, doi: 10.14778/3749646.3749659;Serialtopic:Topic:Acm-Pubtype.

[22] Y. Sun *et al.*, "CGCGraph: Efficient CPU-GPU Co-execution for Concurrent Dynamic Graph Processing," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 3, Sep. 2025, doi: 10.1145/3744904;Journal: TACO

[23] Y. Zhang *et al.*, "LargeGraph: An Efficient Dependency-Aware GPU-Accelerated Large-Scale Graph Processing," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, p. 58, Dec. 2021, doi: 10.1145/3477603;Requestedjournal: TACO;Serialtopic: Acm-Pubtype.

[24] S. Song *et al.*, "Large-Scale Dynamic Graph Processing with Graphic Processing Unit-Accelerated Priority-Driven Differential Scheduling and Operation Reduction," *Appl. Sci. 2025, Vol. 15, Page 3172*, vol. 15, no. 6, p. 3172, Mar. 2025, doi: 10.3390/APP15063172.

[25] J. Zhao *et al.*, "An Efficient ReRAM-based Accelerator for Asynchronous Iterative Graph Processing," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 1, p. 16, Mar. 2025, doi: 10.1145/3689335;Website: Dl-Site; Taxonomy:Acm-Pubtype;Pagegroup:String: Publication.

[26] R. Chen, X. Weng, B. He, and M. Yang, "Large graph processing in the cloud," *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pp. 1123–1126, 2010, doi: 10.1145/1807167.1807297;Topic: Conference-Collections.

[27] "GraphChi | Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation." Accessed: Jul. 27, 2026. [Online]. Available: https://dl.acm.org/doi/10.5555/2387880.2387884

[28] S. Salihoglu and J. Widom, "GPS: A graph processing system," *ACM Int. Conf. Proceeding Ser.*, Jul. 2013, doi: 10.1145/2484838.2484843;Topic:Conference-

Collections.
[29] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *ACM SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, Aug. 2013, doi: 10.1145/2517327.2442530;Serialtopic:Acm-Pubtype.
[30] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *Proc. - Int. Symp. Comput. Archit.*, vol. 13-17-June-2015, pp. 105–117, Jun. 2015, doi: 10.1145/2749469.2750386.
[31] "Gunrock: a high-performance graph processing library on the GPU: ACM SIGPLAN Notices: Vol 51, No 8." Accessed: Jul. 27, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3016078.2851145
[32] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc.*, 2017.
[33] J. L. William L. Hamilton, Rex Ying, "Inductive Representation Learning on Large Graphs," *arXiv:1706.02216*, 2017, doi: https://doi.org/10.48550/arXiv.1706.02216.
[34] P. Veličković, A. Casanova, P. Liò, G. Cucurull, A. Romero, and Y. Bengio, "Graph Attention Networks," *6th Int. Conf. Learn. Represent. ICLR 2018 - Conf. Track Proc.*, Oct. 2017, doi: 10.1007/978-3-031-01587-8_7.
[35] K. Xu, S. Jegelka, W. Hu, and J. Leskovec, "How Powerful are Graph Neural Networks?," *7th Int. Conf. Learn. Represent. ICLR 2019*, Oct. 2018, Accessed: Jul. 27, 2026. [Online]. Available: https://arxiv.org/pdf/1810.00826
[36] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks," *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, pp. 257–266, Aug. 2019, doi: 10.1145/3292500.3330925.