Self-adaptive Graph Traversal on GPUs

Mo Sha
School of Computing
National University of Singapore
sham@comp.nus.edu.sg

Yuchen Li
School of Information System
Singapore Management University
yuchenli@smu.edu.sg

Kian-Lee Tan
School of Computing
National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

GPU's massive computing power offers unprecedented opportunities to enable large graph analysis. Existing studies proposed various preprocessing approaches that convert the input graphs into dedicated structures for GPU-based optimizations. However, these dedicated approaches incur significant preprocessing costs as well as weak programmability to build general graph applications. In this paper, we introduce SAGE, a self-adaptive graph traversal on GPUs, which is free from preprocessing and operates on ubiquitous graph representations directly. We propose *Tiled Partitioning* and *Resident Tile Stealing* to fully exploit the computing power of GPUs in a runtime and self-adaptive manner. We also propose *Sampling-based Reordering* to further optimize the memory efficiency of SAGE through a lightweight and effective node reordering technique on the fly. Extensive experiments demonstrate that SAGE can achieve superior graph traversal performance over existing approaches under different architectural scenarios, i.e., single-GPU, out-of-core, and multi-GPU.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**; **Graph algorithms analysis**; **Parallel computing models**.

KEYWORDS

Graph Processing; GPGPU; Parallel Task Scheduling

ACM Reference Format:
Mo Sha, Yuchen Li, and Kian-Lee Tan. 2021. Self-adaptive Graph Traversal on GPUs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457279

1 INTRODUCTION

Large graphs are pervasive as people and things are digitally connected in a complex way. The information embedded in graphs brings opportunities to discover valuable insights that continuously power the development of data-driven economy. According to Gartner [12], graph processing is becoming a key technology to support decision making in 30% of organizations globally. Such a trend poses an increasing demand for real-time graph processing. In recent years, the rapid evolution of Graph Processing Units (GPUs) has attracted significant attention in both industry and academia as GPUs offer massive computing horsepower for accelerating graph processing workloads [2, 9, 13, 26, 27, 29, 30, 35, 43, 47].

In order to maximize the performance of GPU-based graph processing, existing approaches propose dedicated preprocessing to cater for their GPU-optimized algorithms [3, 11, 23, 31, 37, 38, 50]. Although such approaches can exploit the unique characteristics of the GPU computing paradigm, there are two major drawbacks for practical applications:

- **Preprocessing Overhead.** Existing approaches design sophisticated algorithms to preprocess and transform the original graph representation to dedicated structures. For large graphs with billions of edges, the preprocessing stage can take hours before executing any user workloads. Considering most real-world graph analysis can be processed in a few hours [39], preprocessing incurs significant overheads and cannot be overlooked. Furthermore, when graphs are subject to updates, existing approaches have to rebuild their dedicated structures through the preprocessing stage.
- **Limited Programmability.** Developing new algorithms on the dedicated approaches incurs steep learning curves. In addition to the off-the-shelf algorithms, real-world applications require customized or new graph algorithms. Implementing correct and efficient GPU programs is challenging in general. Furthermore, in large corporations, it is not practical for data engineers to pick the best solution to suit their applications and implement efficient algorithms on the dedicated structures. The issue of programmability magnifies when the application needs to be deployed under different architectures, e.g., single-GPU, out-of-core, and multi-GPU.

Motivated by the aforementioned drawbacks, we propose SAGE, a framework that enables self-adaptive graph traversal to support efficient graph processing on GPUs. SAGE adopts the common node-centric parallel graph processing pipeline and is free from preprocessing, i.e., it starts from the ubiquitous Compressed Sparse Row [45] (CSR) as an initial representation. In other words, after loading the graph data in the CSR format on GPUs, SAGE can immediately respond to queries without any launching latency. As CSR is a widely used graph representation, it also lowers the barrier for data engineers to develop efficient algorithms with SAGE, compared with existing dedicated solutions. Meanwhile, SAGE is self-adaptive and optimized for GPU's hardware characteristics, based on the graph access patterns of the processed workload. By continuously processing the graph on-the-fly, SAGE is able to optimize the GPU efficiency of processing graph data incrementally, and render competitive or even better performance.

To enable SAGE's self-adaptivity, we propose a novel runtime workload reallocation strategy, i.e., *Tiled Partitioning*. In particular, the graph data is dynamically partitioned into tiles, the sizes of which fit the runtime GPU multi-threading resources. Hence, such tiles are suitable to be consumed by the GPU's cooperative

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457279

groups of threads concurrently. The tile accesses will be stored as scheduling logs, i.e., resident tiles, which are then reused when the same data is visited subsequently. Resident tiles avoid the cost of repeated online scheduling and are immediately available for deploying threads to achieve the best performance. Further, resident tiles are visible globally on the device and enable task stealing among all GPU processors, i.e., *Resident Tile Stealing*, to achieve higher parallelism and hence, better hardware utilization.

In addition to the features above, the reusable resident tiles enable profiling statistics to quantify the access patterns of graph data. Hence, we take a step further to propose *Sampling-based Reordering* for better memory access performance. The node indices are adjusted gradually according to the statistics of tile accesses by sampling, in order to increase data locality within tiles, for higher memory efficiency.

The contributions of this paper are summarized as follows:

- We introduce **SAGE**, a GPU-based graph processing framework. We propose *Tiled Partitioning* and *Resident Tile Stealing*, which enable self-adaptivity to harness GPUs' computing power without preprocessing costs.
- We devise *Sampling-based Reordering* to further optimize the memory efficiency when accessing graph data on **SAGE**.
- We conduct an extensive experimental evaluation on different types of graph datasets, which demonstrates that **SAGE** achieves superior performance for applications running on single-GPU, out-of-core, and multi-GPU scenarios.

The rest of this paper is organized as follows. In Section 2, we present the preliminaries. In Section 3, we discuss several challenges related to graph analysis on GPUs and review the existing solutions. In Section 4, we demostrate how **SAGE** performs graph procssing tasks as a framework and provide implementation examples. Section 5 proposes the self-adaptive graph traversal on GPUs, and Section 6 discusses the further optimization of memory access efficiency by *Sampling-based Reordering* on **SAGE**. Next, we present the experimental evaluation in Section 7 and we conclude the paper in Section 8.

2 PRELIMINARIES

2.1 GPU Architecture

A dedicated GPU card is composed of processors and device memory. The advantage of GPUs in accelerating graph computation is attributed to: (a) its massive number of processors; (b) its ultra-high memory bandwidth. The GPU's chip consists of multiple streaming multiprocessors (SMs). Within an SM, the minimum granularity of instruction scheduling is a warp composed of 32 threads, i.e., in each clock cycle, the scheduler can issue an eligible warp (not be stalled) to compute units for executing the next instruction. If the threads of the same warp enter different condition branches, then the scheduler is only able to issue one instruction out of multiple instruction pointers (IP) in each cycle. Hence, the threads that do not belong to the selected IP will be marked as inactive, and the corresponding hardware resources will be idle in this cycle. This aforementioned phenomenon is called warp divergence, which is a major concern of the performance optimization on GPUs [44]. Meanwhile, the two-level cache hierarchy L1 (per SM) - L2 (device) is adopted on GPUs. For boosting the efficiency of parallel

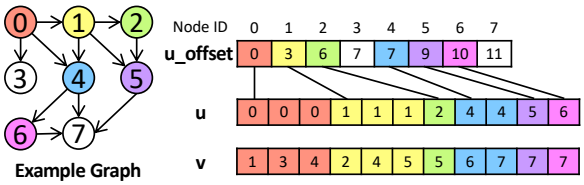

Figure 1: An example of graph representation.

memory access, the corresponding cache line size is designed to be larger than the CPU counterpart, i.e., 128 bytes. Consequently, the uncoalesced memory access will result in a high memory access latency, and cause the effective memory bandwidth to be far lower than the theoretical maximum. This is another major concern of the performance optimization on GPUs [1].

2.2 GPU-accelerated Graph Analysis

Given a graph $G = (V, E)$, V is the node set and E is the edge set. Let $\text{OutDeg}(u) = \{v \in V | (u, v) \in E\}$ denote node u's outdegrees in G. As shown in Figure 1, for the sorted edge list, we can use two arrays (**u** and **v**) to represent the graph data. This representation is called Coordinate Format [36] (COO). For node-centric graph processing, we need to introduce **u_offset** to indicate the range of edges in the edge list, in order to support queries on $\text{OutDeg}(u)$. Using **u_offset** and **v** to represent the graph data is called the Compressed Sparse Row Format [45] (CSR). In this work, we mainly focus on node-centric graph processing based on the CSR format.

In recent years, GPU has attracted a great deal of attention from both academia and industry to explore the usage of GPUs for accelerating graph analysis, such as Breadth-First Search [27, 29, 30, 47, 50], PageRank [9, 14, 33, 35], Connected Component [2, 18, 43], Constrained Shortest Path [28], Graph Pattern Matching [13, 15, 26, 46] and many others. These GPU-optimized algorithms are able to achieve significant performance improvement compared with CPU-based counterparts. Further, some existing studies focus on GPU-based graph processing in specific application scenarios, e.g., dynamic graphs [40, 51] and compressed graphs [41].

Meanwhile, some existing studies focus on general GPU-based graph computing platforms. Medusa [53] is a pioneer and proposes a set of user-defined APIs for graph computation. Users can implement graph algorithms in a sequential style and the generated codes can be executed effectively on GPUs. Gunrock [48] provides a more flexible design to support a broader range of graph applications and the graph computation can be conducted on multiple GPUs. Groute [3] presents an asynchronous graph computational model specific to the single node multi-GPU scenario in order to improve the hardware utilization. Lux [19] further supports the distributed multi-GPU system. In a nutshell, these platforms take the GPU's hardware characteristics into full consideration and design universal APIs for graph computation. The user-customized graph algorithms can be executed efficiently on such platforms without careful consideration for the GPU's hardware characteristics.

3 CHALLENGES AND RELATED STUDIES

In this section, we discuss several challenges on how to achieve superior performance for GPU-accelerated graph processing, and we review how existing studies address each challenge.

3.1 Parallelism and Load Balancing

In node-centric parallel graph processing, the parallel tasks are scheduled at the granularity of frontiers (i.e., nodes). It is widely known that for most graph data, the outdegree distribution of nodes is highly skewed, as large graphs often follow a power-law distribution. Therefore, to process frontiers in parallel, the load distribution of scheduled tasks is imbalanced, and such load imbalance will degrade the performance. Furthermore, the GPU issues a warp (32 threads) in the SIMT manner. Hence, if the frontiers (corresponding to the threads in the same warp) do not have the same |outdegree|, the hardware resources cannot be fully utilized due to warp divergence. To alleviate the low parallelism issue due to load imbalance, existing work proposes solutions from two perspectives [32]: (a) online thread resource reallocation, (b) graph preprocessing.

Among the existing work, D. Merrill et al. [30] propose an effective online resource rescheduling. The main idea is to schedule the hardware resources of blocks, warps, and threads for frontiers with different loads. Such an online task reallocation depends on the inter-thread communication and the synchronization mechanism, and each thread will synchronize and access the predefined shared memory periodically in order to be aware of the collaboration requests from other threads. However, it introduces considerable synchronization overheads and is not benefical to iner-SM load imbalance due to the device limitation. Meanwhile, to alleviate the irregularity of the original graph data distribution, the preprocessing techniques [6, 21, 37] on graph representations are proposed. An example is `Tigr` [37], which introduces auxiliary structures, and specifically conducts a partition on the edges of the nodes with a large |outdegree| in order to migitate the effect of their outdegree distribution. However, we note that the graph topology is actually altered in this process. Therefore, to achieve equivalent results on the precessed graph, application-specific adjustments are required. This may bring a huge challenge when developers implement customized graph applications.

3.2 Cache and Memory Efficiency

GPU-based graph analysis is generally memory-intensive, and the performance of GPU-based graph analysis cannot be easily estimated by the theoretical memory bandwidth of GPUs. If the memory access patterns of the application lack locality (i.e., random-like access), the effective memory bandwidth will be lowered significantly. The main reasons are two-fold: (a) under random access, the hit rate of caches is rather low; (b) in a cache miss, the memory controller will use the cache line as the granularity to swap data between memory and cache. This causes memory access amplification, which can be measured as a ratio between the effective memory bandwidth and the actual memory bandwidth. Unfortunately, graph data generally does not exhibit a high locality and the size of the GPU cache line is as large as 128 bytes. Take 4-byte labels as an example, if the neighbors' indices are scattered, the access amplification can reach 32x in the worst case.

For boosting the memory/cache efficiency in graph analysis, existing work mainly employs the reordering technique. Node reordering is based on a bijection $\sigma : V \rightarrow V$, which assigns a new labelling to all nodes in the graph, to improve the locality of outdegrees' indices. Existing studies propose different approaches based on various cost models, to improve the memory efficiency of graph

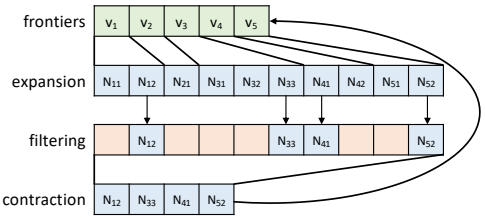

Figure 2: Parallel graph processing computational pipeline.

representations. Reversed Cuthill–McKee [10] (`RCM`) reduces the index range of a node's outdegrees through reducing the bandwidth of sparse adjacency matrix. Layered Label Propagation [5] (`LLP`) enables the nodes within the same clustering to have contiguous indices via graph clustering. Further, `Gorder` [49] maximizes the window-based sum of common indegrees to achieve superior performance.

Although the reordering-based approaches manage to improve the locality of neighbors' indices based on the graph's topology, the computational pipeline of parallel graph processing is not taken into consideration. `CuSha` [23] proposes to optimize towards the computational pipeline of GPU-based parallel graph processing, and employs preprocessing techniques to divide the outdegrees into shards according to their indices. Nonetheless, this approach assumes that the parallelism is high enough, i.e., for any shard (i.e., an adjacent index interval), there are always enough neighbors to be processed so that the GPU can be fully utilized.

3.3 Memory Access in Out-of-Core Scenarios

In the out-of-core scenarios, we need to access data that is not stored on device memory. One strategy is to access data in the external memory on-demand. The performance challenge of this on-demand access lies in that the communication frame consists of both control segment (header) and data segment (data payload) no matter which communication protocol is used. Graph analysis tends to generate massive and non-contiguous access requests, which leads to massive of communication frames with a small payload. The requests cause the controller to be under high pressure, and the ratio of the effective payload is thus reduced, which brings down the effective bandwidth to a large extent. The other strategy is to maintain an out-of-core data pool in the local device memory in a cache-like manner, e.g., unified memory [25] (UM).

In order to improve the efficiency of external memory access in GPU-based graph analysis, some techniques are proposed. S.W. Min et al. [31] demonstrate the detailed behaviors of accessing the host memory through PCIe and propose to maximize the PCIe bandwidth via "merged" and "aligned" access behaviors. `HALO` [11] suggests a reordering-based optimization approach to boost the efficiency of UM in graph applications in which the nodes are ordered according to their centrality in the graph topology. Another example is `Subway` [38] in which a subgraph extraction approach is presented to identify the active edges in the current situation, and then the desired data in the external memory is preloaded in an asynchronous way.

4 GRAPH PROCESSING FRAMEWORK

The parallel graph processing framework of `SAGE` falls into the node-centric processing paradigm, and adopts the widely employed

Algorithm 1 `filtering` of Example Applications

```
procedure filter(frontier, neighbor)
  // for Breadth-First Search
  if dist[neighbor] = -1 then
    dist[neighbor] := dist[frontier] + 1
    filter_in(neighbor)
  end_if

  // for Betweenness Centrality (forward phase)
  if dist[neighbor] = -1 then
    d := atomicCAS(dist[neighbor], dist[frontier] + 1)
    if d = -1 then
      filter_in(neighbor)
    end_if
  end_if
  if dist[neighbor] = dist[frontier] + 1 then
    atomicAdd(sigma[neighbor], sigma[frontier])
  end_if

  // for Betweenness Centrality (backward phase)
  if level[neighbor] = level[frontier] + 1 then
    increment := sigma[frontier] / sigma[neighbor]
    increment *= delta[neighbor] + 1
    atomicAdd(delta[frontier], increment)
  end_if

  // for PageRank
  increment := pr_in[frontier] * 0.85
  increment /= outdegree_cnt[frontiter]
  atomicAdd(pr_out[neighbor], increment)
end_procedure
```

pipeline based on double-buffering frontier queues, i.e., iteratively executing a pipeline of **expansion** - **filtering** - **contraction** as shown in Figure 2. Each iteration starts from the frontier array that denotes the active nodes in the current iteration. First, in the **expansion** step, the outdegrees of all frontier nodes are expanded. Next, in the **filtering** step, the outdegrees expanded in the expansion step (i.e., the neighbors of the active nodes in the current iteration) will be investigated if they should pass the filter and enter the next iteration. Finally, in the **contraction** step, the unfiltered neighbors are compressed to a contiguous array and used as the frontiers in the next iteration. It is clear that in such a computational pipeline, it is easy, in each step, to parallelize the workloads and execute them in a multi-threaded manner.

The above pipeline that SAGE adopts can effectively support a wide range of graph applications. Essentially, through accessing and updating the attributes of nodes, the computational pipeline decides if a node will be active in the next iteration. This iterative process is terminated when there are no frontiers generated, meaning that the attributes of all nodes (i.e., the desired outputs of this application) are converged according to the requirements of the application. The main logic of graph applications is implemented in the filtering step, in which edges between frontiers and their neighbors are traversed. Therefore, developers only need to implement the filtering interface to customize their applications.

Algorithm 1 shows the implementation of three representative graph algorithms, i.e., Breadth–First Search (BFS), Betweenness Centrality (BC), and PageRank (PR), based on SAGE. Taking BFS as an example, in the filtering step, each neighbor will be checked if it has already been traversed via accessing the **distance** attribute. If not, the **distance** attribute is updated and the corresponding neighbor can pass the filter and will be the frontier of the next iteration. Due to limited space, we only elaborate on three graph applications as examples and evaluate them in the experiments. However,

Algorithm 2 Load Reallocation by Tiled Partitions

```
procedure expandFrontiers(frontiers[], csr)
  frontier := frontiers[thread_id]
  u_beg := csr.u_offsets[frontier]
  u_end := csr.u_offsets[frontier + 1]
  neighbor_size := u_end - u_beg

  tile := cg::this_thread_block()
  while tile.size() ≥ MIN_TILE_SIZE do
    while tile.any(neighbor_size ≥ tile.size()) do
      leader = tile.elect(neighbor_size ≥ tile.size())

      gather := tile.shfl(u_beg, leader)
      gather += tile.thread_rank()
      if leader = tile.thread_rank() then
        neighbor_size %= tile.size()
        u_beg := u_end - neighbor_size
      end_if
      gather_end := tile.shfl(u_end, leader)
      leader_frontier = tile.shfl(frontier, leader)

      while (tile.all(gather < gather_end)) do
        neighbor := csr.v[gather]
        filter(tile, leader_frontier, neighbor)
        gather += tile.size()
      end_while
    end_while

    tile := cg::partition(tile)
  end_while

  cg::this_thread_block().sync()
  handle_fragment(frontier, u_beg, u_end, csr)
end_procedure
```

the pipeline based on iterative graph traversal, which is adopted by SAGE, has been employed to support a wide range of graph processing primitives through customized `filter(frontier, neighbor)` interfaces, including but not limited to: (a) Tarjan (construct the search tree and propagate the lowest timestamps); (b) Connected Component (merge two components of the frontier and the neighbor on the disjoint-set forest); (c) Label Propagation (identify the label majority among all neighbors of a frontier); (d) Shortest Path (iteratively update neighbors' distances); etc.

5 SELF-ADAPTIVE GRAPH TRAVERSAL

In this section, we propose how SAGE conducts a self-adaptive graph traversal without a need for preprocessing.

5.1 Load Reallocation by Tiled Partitions

SAGE starts from CSR where the outdegree distribution can be seriously skewed to cause load imbalance. Hence, we conduct load reallocation in the runtime. In the beginning of every graph processing iteration, each frontier (i.e., active node) is assigned to a thread. Inspired by [30], a thread will, according to the |outdegrees| of the corresponding frontier assigned, preempt other threads and distribute its neighbors to them for processing. Different from [30] that decides if the task of a certain frontier should be handled by a block, a warp or a thread, based on |outdegrees| and the GPU's thread number in different thread hierarchical groups, we propose a more fine-grained load reallocation, i.e., *Tiled Partitioning*, based on tiled partitions when expanding the outdegrees of a frontier. Throughout this paper, a **tile** refers to a group of threads in a collaborative state, i.e., a particular number of threads functioning as one by communicating closely and executing synchronously.

Algorithm 2 illustrates the expansion step of the node-centric parallel graph processing shown in Figure 2, with a *Tiled Partitioning* manner. The input is the frontier array and the graph data represented in CSR format. In this step, we conduct the filtering step (highlighted in line 23) for each expanded outdegree of each node in the frontier array.

• **Thread Management:** We note that Algorithm 2 is executed in parallel in the SIMT manner. That is, each thread executes the same program and distinguishes itself from others via `thread_id`. For instance, in lines 2-5, each thread gets the frontier (`node`) assigned to it via `thread_id` and obtains the range of the outdegrees of `node` in `csr.v` from `csr.u_offsets`. Further, each thread gets its |outdegrees|, i.e., `neighbor_size` for load reallocation. The functions highlighted in purple in the pseudocode are the interfaces of Cooperative Group [16] (CG). CG represents a group of threads in a collaborative state, and the threads of one CG cooperate via calling APIs of the same instance of CG they hold. In other words, similar to `thread_id`, threads distinguish the cooperative thread groups by different instances of CG they hold in the SIMT execution. Through `this_thread_block()`, the threads belonging to the same block will get the same instance of CG, i.e., `tile`. In the following discussion, `tile` refers to an instance of CG that is held by a group of threads in a collaborative state.

• **Leader Election:** In line 7, `tile` is initialized as the threads in the whole block, i.e., `tile.size()` is the thread number in a block. After that, in the loop of lines 9-26, if a thread's current |outdegrees| exceeds `tile.size()`, it joins the election to become the leader so that all threads in the whole `tile` consumes the leader's workloads collaboratively, until there are no more threads with a |outdegrees| larger than `tile.size()`. `any(bool)` and `all(bool)` are vote functions of CG, and return a Boolean variable denoting if any/all of the conditions passed from all threads of the `tile` are true. Lines 10-19 describe the leadership election and the process in which the leader thread assigns its loads to `tile` for collaboration. In detail, among all threads that input a true condition, `elect(bool)` returns one of their ranks. `shfl(variable,rank)` is the shuffle function that returns `variable` value of the corresponding thread.

• **Thread Cooperation:** When the leader is elected in line 10, in lines 12-13, each thread in `tile` gets the pointer of the outdegree to handle, according to the beginning offset of the outdegrees to be processed by the leader, as well as its own rank in `tile`. In lines 14-17, the leader thread updates its outdegree interval to process, as these tasks are already distributed to `tile`. Lines 21-25 depict how the `tile` processes the leader's outdegrees in a collaborative and parallel manner. Specifically, in line 22, each thread reads the index (v) of the corresponding outdegree from CSR and then, in line 23, conducts the following filtering step on the neighbor node. When the execution reaches line 28, it means that the condition in line 9 turns false, i.e., there are no more threads in `tile` with the number of outdegrees to handle larger than the tile size. Then the tile splits into multiple smaller CGs that continue to handle the threads' outdegrees to be processed.

• **Example:** Figure 3 illustrates an example of load reallocation by tiled partitioning. For simplicity, we assume a block consists of 16 threads in the figure. Further, as for tiled partitions, we adopt the

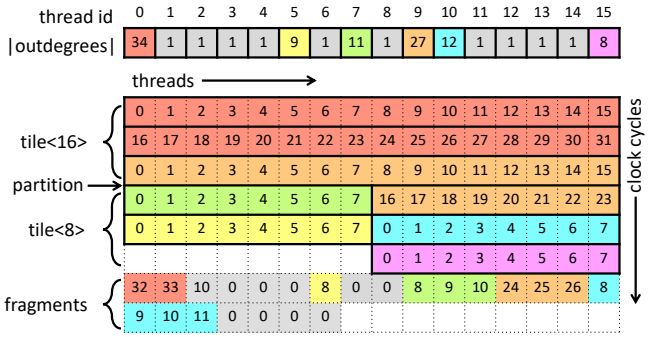

Figure 3: An example of tiled partitioning.

binary partition and set `MIN_TILE_SIZE=8`. Each of the 16 threads in this block is assigned one frontier, with each frontier's |outdegrees| specified in the grids on the top. Some of the frontiers have a larger |outdegrees| and are highlighted in bright colors, whereas the rest of the frontiers in grey have only one outdegree. In the beginning (line 7), all 16 threads in the block hold the same instance of `tile` with a size of 16, i.e., `tile<16>`. Therefore, the red thread and the orange thread will join the election in line 10, due to their `neighbor_size` (i.e., 34, 27) being larger than `tile.size()` (i.e., 16). Suppose the red thread's `u_beg` and `u_end` are [0, 34) and it wins the first election. Then in lines 12-13, each thread obtains the red thread's `u_beg`, adds its own `thread_rank()` to derive the offset of its corresponding outdegree to process in collaboration. In lines 14-17, the red thread's `u_beg` is adjusted to 32, as it has distributed the tasks of its 0-31 outdegrees to other threads. Next, the loop in lines 21-25 will be executed twice, and each time the 16 threads in `tile` will handle the 16 outdegrees of the red thread. When the execution reaches line 21 for the third time, the remaining outdegrees of the red thread are fewer than 16, i.e., some threads' `gather` exceeds `gather_end` and hence, the loop ends. Similarly, the orange thread wins the second election in line 10 (as there are no other candidate threads), and 16 out of its 27 outdegrees will be consumed. After that, the tile will be partitioned. To be specific, when the execution of all the 16 threads reaches line 28, threads 0-7 will get an instance of `tile<8>`, while threads 8-15 will get another one. Then these two groups of threads do not communicate with each other anymore, and each group handles the frontiers with `neighbor_size` larger than 8 collaboratively within the group. For example, `tile<8>` composed of threads 0-7 will in order handle the 8 outdegrees out of the remaining 11 ones held by the green thread, and then the 8 out of 9 held by the yellow thread. When `tile` is partitioned into a smaller size than `MIN_TILE_SIZE` (line 8), the task fragments (i.e., remaining neighbors) of each thread are no longer considered imbalanced.

We skip the implementation details of handling fragments (line 32), and refer interested readers to the fine-grained scan-based gathering discussed in [30]. Grouping multiple threads in tiled partitions in a dynamic manner can further improve the usage of parallel computational resources of GPUs, when the graph data distribution is imbalanced and irregular. We note that limited by the scale of the example, this advantage may not be sufficiently demonstrated, as the size of a block can generally be up to one thousand.

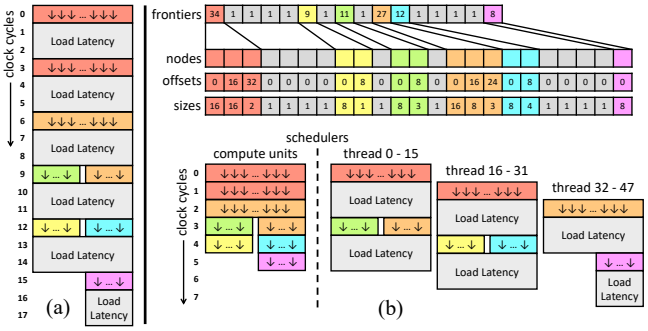

Figure 4: An example of work stealing by resident tiles.

Algorithm 3 Resident Tile Stealing

```
1  procedure expandFrontiers(frontiers[], csr)
2    // expand the tiled partitions of the frontier
3    nodes, offsets, sizes = expandTiles(frontiers, csr)
4    node := nodes[thread_id]
5    u_beg := offsets[thread_id]
6    neighbor_size := sizes[thread_id]
7    u_end := u_beg + neighbor_size
8
9    // consume the partitions with a corresponding size
10   tile := cg::this_thread_block()
11   while tile.size() ≥ MIN_TILE_SIZE do
12     while tile.any(neighbor_size = tile.size()) do
13       leader = tile.elect(neighbor_size = tile.size())
14       // similar to Algorithm 2
15     end_while
16     tile := cg::partition(tile)
17   end_while
18
19   cg::this_thread_block().sync()
20   handle_fragment(u_beg, u_end, csr)
21 end_procedure
```

5.2 Work Stealing by Resident Tiles

The approach based on task reallocation to solve the load imbalance when expanding frontiers is essentially to split a frontier's outdegrees into multiple tiled partitions with specific sizes, and then handle each partition in parallel through adjusting the size of CG dynamically. Though this approach can improve the hardware utilization and parallelism when processing the imbalanced graph data, there exist several major disadvantages. First, the dynamic adjustment in the runtime will inevitably introduce non-negligible overheads, which restricts the maximum performance. Second, limited by the hardware design of GPUs, such runtime adjustments can only be executed within a block. This means that such adjustments can only function within an SM, but cannot benefit the load imbalance among SMs. Third, in a collaborative state, though each thread is assigned several tasks, these tasks will not enter the execution pipeline until the thread is elected leader in the thread group. This characteristic will degrade the potential parallelism. For GPU-based graph analysis, as discussed in Section 3.2, the graph computation is generally memory-intensive, meaning that for a single thread, most instruction cycles stall due to the memory latency. As a result, we need the parallelism to be high enough to fully occupy the memory pipeline.

Following the example in Figure 3, as illustrated in Figure 4 (a), when the red thread wins the control in `tile` by election, in line 22, the 16 threads in `tile` read the index of the outdegree from CSR, and the whole `tile` will stall until the end of memory access (generally hundreds of cycles). However, if there are no (or not enough) other threads of the SM waiting to be scheduled, the scheduler has no instructions to issue in the next cycle and both the compute units and the memory pipe are wasted. In other words, the tasks of one `tile` cannot be used to amortize the memory access latency, as they are executed in tiles sequentially.

Motivated by the drawbacks of runtime reallocation, we propose to keep the thread arrangement in the GPU's device memory to avoid repeated dynamic scheduling, and achieve more effective work stealing. In particular, we adopt a hybrid strategy, i.e., when consuming the frontier array and expanding the corresponding outdegrees in a multi-threaded manner, we employ the techniques described in Section 5.1 to split the long outdegree array to tiled partitions with a specific size as well as suitable to the configuration of the GPU's thread group. Further, we keep the online task reallocation results, i.e., the tiled partitions of a frontier, in the GPU's device memory, as the running context of an auxiliary structure. Hence, when we revisit the outdegrees of a certain node, there is no need to arrange them dynamically again.

When expanding frontiers, we conduct two steps, as shown in Algorithm 3. First, we expand the tiled partitions of the frontier in the GPU's device memory (lines 2-7). Second, similar to Algorithm 2, CGs consume the tiles with a corresponding size from the tiled partition array, and adjust the size of CGs through continuous partitioning until termination (lines 9-17). We reuse the example in Figure 3 for explanation, and the details of the aforementioned process are illustrated in Figure 4 (b). There are still 16 threads corresponding to 16 frontiers, but this time, we first expand the tiles of the frontiers to the device memory. Next, all threads form CGs in blocks, and start to consume tiles from the tiled partition array and partition themselves continuously. Because the tiled partitions are expanded to the device memory, they are visible to the whole GPU. Thus, any CGs with an appropriate size, rather than only the one corresponding to the frontier, can consume the tiled partitions. It is clear that compared with the case shown in Figure 4 (a), this is equivalent to flattening the tiled partitions to process in advance (as discussed above) and making them visible to the whole device. Then the aforementioned problems of SM load imbalance and memory pipe not being fully occupied can be solved.

5.3 Discussions

A GPU consists of a large number of simplified cores with poor single-threaded performance and conditional branching capability. When dealing with the sparsity and the skewed distribution of large graphs, an imbalanced load caused by coarse-grained workload mapping leads to amplified performance loss. Therefore, the major concerns to achieving efficient parallel graph processing are how to arrange the irregular graph traversal into independent subtasks and schedule these subtasks to multiple processors. In this section, we mainly focus on how **SAGE** deals with these two concerns, that is, the proposed *Tiled Partitioning* and *Resident Tile Stealing*, respectively. To showcase the benefits from the design of **SAGE**, we compare it with approaches proposed by existing studies.

`Tigr` [37] preprocesses graphs data into uniform-degree tree transformation (UDT), which reduces the skewed distribution of graphs by introducing intermediate nodes to split large outdegrees

Algorithm 4 Tile Access Sampling

```
1  procedure filter(tile, frontier, neighbor)
2    __shared__ tile_neighbors[]
3    tile_neighbors[tile.thread_rank()] := neighbor
4    tile.sync()
5    my_sector = neighhor / SECTOR_WIDE
6    cnt := 0
7    for i in tile.size() do
8      if tile_neighbors[i] / SECTOR_WIDE = my_sector then
9        cnt++
10     end_if
11   end_for
12   locality[frontier] += cnt
13   // application specific filtering implementation
14 end_procedure
```

based on fixed cutpoints. In addition to the overhead introduced by preprocessing and auxiliary structures, a predefined degree splitting rule needs a case-by-case fine-tuning for different hardware and graphs. `B40C` [30] classifies frontiers into three buckets based on the number of outdegrees and then handles frontiers in each bucket separately. Therefore, tasks in different buckets can be processed by three different concurrency schemes configured in advance, i.e., letting larger thread groups handle frontiers with larger outdegrees and vice versa. The thread task rescheduling relies on synchronizations, and can only steal workloads in the same SM due to the device limitation.

In the proposed `SAGE`'s *Tiled Partitioning*, it does not first partition tasks by predefined strategies as in most existing studies. Instead, it dynamically adjusts the sizes of thread groups (from large to small) in the runtime and consumes a part of workloads only when it can fully utilize the threads. In this process, irregular loads are consumed in a fine-grained manner and high utilization is ensured. Meanwhile, the preprocessing-free scheduling mechanisms should be conducted in the runtime. Such online scheduling for high-quality load balancing tends to rely heavily on conditional branching and synchronization, which are also very costly on GPUs. Hence, we propose a lightweight *Resident Tile Stealing* mechanism to keep the intermediate results of tiled partitioning in the device memory and the intermediate results are used for runtime scheduling. We include `Tigr` and `B40C` as baselines in the evaluation, and the benefits of `SAGE`'s design on task partitioning and stealing are empirically validated in Section 7.

Further, in `SAGE`, the memory access of graph processing is conducted by thread groups with specific sizes, i.e., concurrent tile access, which provides opportunities for further optimizations. Particularly, we apply a tile alignment strategy, i.e., making tiled partitions aligned with the physical memory sectors to optimize the concurrent memory access. In Section 6, we also propose *Sampling-based Reordering* to further improve the memory efficiency.

6 SAMPLING-BASED REORDERING

In Section 5, we mainly discuss the expansion step shown in Figure 2. In this section, we focus on the next step after the outdegrees of frontiers are expanded, i.e., the filtering step, and discuss its performance concerns. In the filtering step, we need to decide if the current neighbor node will become an active node, i.e., the frontier in the next iteration, through accessing and updating the value of the node, e.g., the visited label of BFS. The performance concern mainly lies in the memory/cache performance degradation due to the lack of locality in the outdegree index distribution, as discussed in Section 3.2. Most existing approaches are based on reordering-based graph preprocessing, i.e., to generate a new replica of the graph by reassigning indices to the nodes with a bijection mapping beforehand, and then conduct the graph analysis on the reordered replica to improve the locality of the outdegree indices and boost the access efficiency. As for the preprocessing, the actual memory access behaviors are hard to be known in advance; hence, reordering-based preprocessing approaches have to make certain assumptions about the actual memory access behaviors and build cost models to preprocess the graph data.

In `SAGE`, we propose the *Sampling-based Reordering* that optimizes the node indices for higher memory access locality; nonetheless, `SAGE` adjusts the node indices in the runtime instead of relying on preprocessing. Based on the discussion in Section 5, `SAGE` handles the node outdegree array based on tiled partitions and keeps them in memory for reuse so that the access patterns that adapt to the current hardware are determined throughout the runtime. Hence, we can quantify the access behaviors to the graph representation more directly and precisely, and optimize the memory efficiency by adjusting the node indices. In `SAGE`, the reads on the graph data from graph applications are concurrent memory access in tiles. For `tile<m>` that executes reads on the data of consecutive m outdegrees as neighbors of the frontier, the actual costs of memory reads correspond to the number of sectors that are occupied by the memory reads, i.e.,

$$\text{count}\left(\text{distinct}\left(\left\{\left\lfloor \frac{\text{neighbors of tile}}{\text{sector\_wide}} \right\rfloor\right\}\right)\right)$$

As for the memory accesses specific to a certain graph application scenario, given T as the collection of all tiles accessed and a bijection $\sigma : V \to V$, the reordering that achieves the optimal memory efficiency, i.e., the minimum sector access is

$$\arg\min_{\sigma} \sum_{\text{tile}\in T} \text{count}\left(\text{distinct}\left(\left\{\left\lfloor \frac{\sigma(\text{neighbors of tile})}{\text{sector\_wide}} \right\rfloor\right\}\right)\right)$$

Theorem 6.1. *Calculating the permutation σ that achieves the minimum sector accesses is NP-hard.*

Proof. We construct a special case that can be reduced to a minimum linear arrangement (MLA) problem with binary distancing. Given a graph $G = (V, E)$, for each $(u, v) \in E$, we construct a `tile<2>` including u and v. Let k = sector_wide. This case is then equivalent to solve:

$$\arg\min_{\sigma} \sum_{(u,v)\in E} f(|\sigma(u) - \sigma(v)|), \quad f(x) = \begin{cases} 0 & \text{if x < k,} \\ 1 & \text{otherwise.} \end{cases}$$

MLA is a classic NP-hard graph problem [20] and its variant of binary distancing with a finite k is proved to be equivalent to the original MLA [34]. □

Due to NP-hardness, we propose a heuristic, i.e., *Sampling-based Reordering* to derive the permutation. Algorithm 4 demonstrates the main idea that we sample the tile access and for each time of tile access, the number of intra-tile neighbors that are located in the same memory sector with each other is counted as the locality measurement. Since the sampling mainly operates within shared memory, the cost is lightweight. Based on the access statistics of

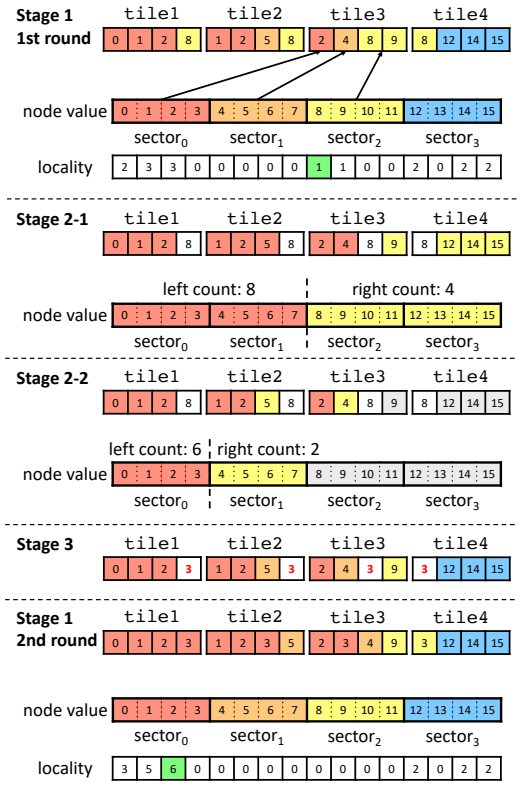

Figure 5: An example of sampling-based reordering.

a tile, we try to search a new index for the tile, which enables the node to have more intra-tile neighbors within the same sector. SAGE's sampling-based memory access optimization by reordering consists of three stages. In each stage, we will sample tile access to collect the corresponding statistics in order to conduct the reordering. In particular:

- **Stage 1**: for each node, calculate the number of intra-tile nodes that fall into the same sector, as the measure of locality.
- **Stage 2**: find a potentially better index for each node by binary search, which may lead to higher locality.
- **Stage 3**: calculate the actual locality changes based on the new index obtained in Stage2.

Each time when Stage 3 is finished, we will update the indices of the nodes whose localities are increased, in the graph representation. We repeat this process of three stages round by round, and optimize the indices of the nodes according to the actual memory access behaviors, so that the memory efficiency can be boosted gradually until convergence to a relatively high level.

We illustrate an example in Figure 5 on how to collect statistics of the memory access behaviors through investigating the nodes that are processed by `tile` concurrently in the filtering step, and then improve memory efficiency by reordering. In this example, suppose that in each stage, the memory accesses are the same, i.e., four `tile<4>`, and one sector can contain values of four nodes.

Stage 1: In Figure 5 Stage 1, we sample the access consisting of 4 `tile<4>` that read values of the corresponding neighbors in tiles. For instance, when **tile3** is accessed, although only 4 nodes' values are needed, 12 nodes' values are actually loaded from memory,

Table 1: Statistics of Datasets

Dataset	Category	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
`uk-2002`	Web	18.5M	298M	16.1
`brain`	Biology	784K	267M	683
`ljournal`	Social Network	5.3M	79M	14.9
`twitter`	Social Network	41.6M	1.46B	35.1
`friendster`	Social Network	65.6M	1.81B	27.5

since the desired values fall in 3 sectors ($\text{sector}_{0,1,2}$). Thereby, we increase the localities of **node-8** by "1", meaning that we count the event that one node (**node-9**) within a tile access loads the same sector (in yellow). According to tile access in this stage, for each node, we hereby count the total number of intra-tile neighbors that are in the same memory sector, as the locality measure of its index.

Stage 2: When the sampled tile access in Stage1 reaches a predefined sampling threshold, we will turn to Stage2. In this stage, for each node, we try to search a better index so that if the node uses this new index, the intra-tile locality can be increased compared with that in Stage1. We look for potentially better new indices by binary search. Specifically, starting from the whole interval, for each step of the binary search, we sample and measure which half area has more intra-tile nodes till the searching range converges to one sector. In this example, we only consider **node-8** for simplicity. In Figure 5 Stage 2-1, the first step of the binary search, the search area is initialized to [0, 15]. We first evaluate which half area ([0, 7] in red or [8, 15] in yellow) has more intra-tile nodes of **node-8**. Then, in Stage 2-2, the binary search completes, indicating that sector_0 is a potentially better index for **node-8**.

Stage 3: Similar to Stage 1, we measure the actual locality of the new index obtained in Stage 2. In the example, if **node-8**'s index is changed to **3**, the locality of **node-8** will increase from **1** to **6**. According to the locality comparison between two indices calculated in Stage 1 and Stage 3, we decide whether to change the index of the current node and update the graph representation.

The complexity of this reordering consists of two parts, namely sampling stages, and updating the graph representation after Stage 3. The sampling is performed along with the tile access, that is, if we collect T tiles in each stage, the complexity is $O(\log|V| \cdot |T|)$. We note that after Stage 3, we will have an array of the expected index of each node, which could contain duplicated index values of different nodes or have discontinuous index values. We then sort the expected index array to determine the actual index order. To update the graph representation, we adopt `bb_segsort` [17] for the efficient index replacement on the GPU. Sorting is a GPU-friendly primitive that can be performed efficiently by radix sort in parallel with the complexity $O(k \cdot n)$, where k is considered as a constant. Hence, the complexity of this updating is $O(|V| + |E|)$.

7 EXPERIMENTAL EVALUATION

In this section, we demonstrate the empirical performance of our proposed SAGE. In Section 7.1, we introduce the experimental setup. In Section 7.2, we compare our proposal with different baselines in three representative architecture scenarios in order to validate the effectiveness and generalizability of SAGE. Further, in Section 7.3, we investigate the internal mechanisms of SAGE and conduct an ablation study to verify the impact of our proposed optimizations.

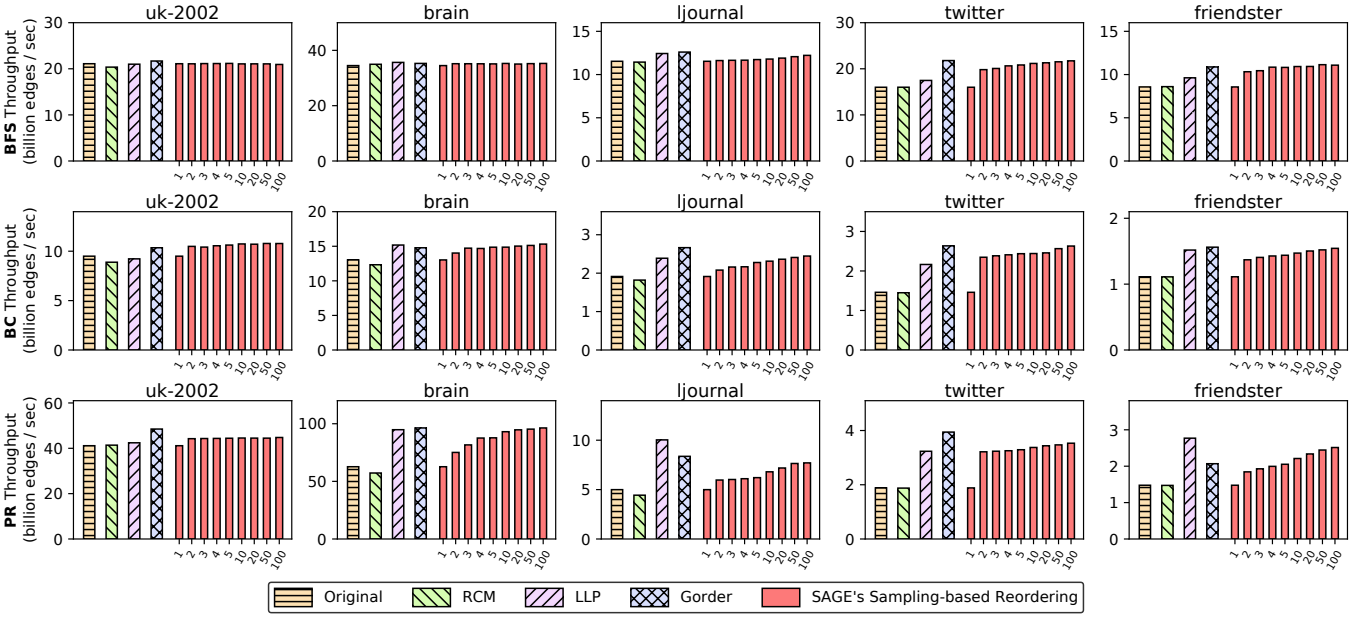

Figure 6: Comparison between SAGE and reordering methods.

7.1 Experimental Setup

Datasets. We collect five real-world graph datasets of different categories and in different scales for experimental evaluation. The detailed statistics of these datasets are summarized in Table 1.

- `uk-2002` is a web graph dataset collected in 2002. It is crawled by UbiCrawler [4] from the web pages that belong to .uk domain.
- `brain` is a biology graph dataset that records the link structure between neurons of human beings' brain. This dataset is collected by NeuroData[1] and provided by NetworkRepository[2].
- `ljournal` is a social network graph dataset that records the friendship relationship of LiveJournal[3]. It is a free online blog service that involves millions of users. This dataset is a snapshot collected in 2008 [7].
- `twitter` contains the relationship between users and followers in Twitter[4]. This dataset is a snapshot collected through Twitter API by [24] in 2010.
- `friendster` is a dataset collected by [52]. This dataset records the friend relationship of a social and online gaming network[5].

Baselines. We compare SAGE with existing reordering methods and parallel graph processing (PGP) approaches.

Reordering Methods:

- `RCM` [10]: A permutation proposed to reduce the bandwidth of an adjacency matrix, as a variant of Cuthill-McKee algorithm [8].
- `LLP` [5]: A permutation determined by the layered label propagation algorithm.
- `Gorder` [49]: A permutation given based on a defined locality score *Gscore* that is maximized by greedy strategies and partial maxTSP calculation on sliding windows.

PGP Approaches:

- `Ligra` [42]: The state-of-the-art CPU-based parallel graph analytic framework for NUMA-based multiprocessor machine.
- `Tigr` [37]: A novel graph preprocessing solution of irregular graph transformation for GPU-friendly graph processing.
- `Gunrock` [48]: A novel GPU-based graph processing platform.
- `Groute` [3]: A novel GPU-based asynchronous framework tailored for multi-GPU graph processing.
- `Subway` [38]: A novel GPU-based framework tailored for out-of-GPU-memory graph processing.
- `B40C` [30]: A novel graph traversal method on GPUs, which introduces three predefined strategies in order to handle nodes with different scales of neighbors.
- `SAGE`: The proposed self-adaptive graph traversal on GPUs.

Experimental Environment. The experimental evaluation results are conducted on a GPU Server equipped with 2 × Intel Xeon Gold 6140 CPUs (2.3Ghz, 36 cores), 384 GB main memory, and 2 × NVIDIA QUADRO RTX 8000 GPUs (4608 cores, 48 GB device memory). All source codes are compiled by GCC-7.5 and CUDA 11.0 in C++14 standard with -O3 under Ubuntu 18.04.4 LTS. OPENMP is used to support parallel primitives for `Ligra`. NVIDIA Nsight Compute 2020.1 is used as the profiling tool of GPU's kernels.

7.2 Comparison Result Analysis

We present the experimental results between SAGE and the baseline methods introduced in Section 7.1, and then analyze and discuss the results according to different architectural scenarios: single-GPU, out-of-core, and multi-GPU. We demonstrate an extensive evaluation in the single-GPU scenario in three graph applications (i.e., BFS, BC, and PR), and only discuss BFS performance for the other scenarios due to limited space and their orthogonality. For BFS and BC, the performance is measured by randomly selected source nodes. All experiments are repeated 100 times to calculate

[1] https://neurodata.io/data/
[2] http://networkrepository.com/bn-human-Jung2015-M87113878.php
[3] https://www.livejournal.com/
[4] https://twitter.com/
[5] http://www.friendster.com

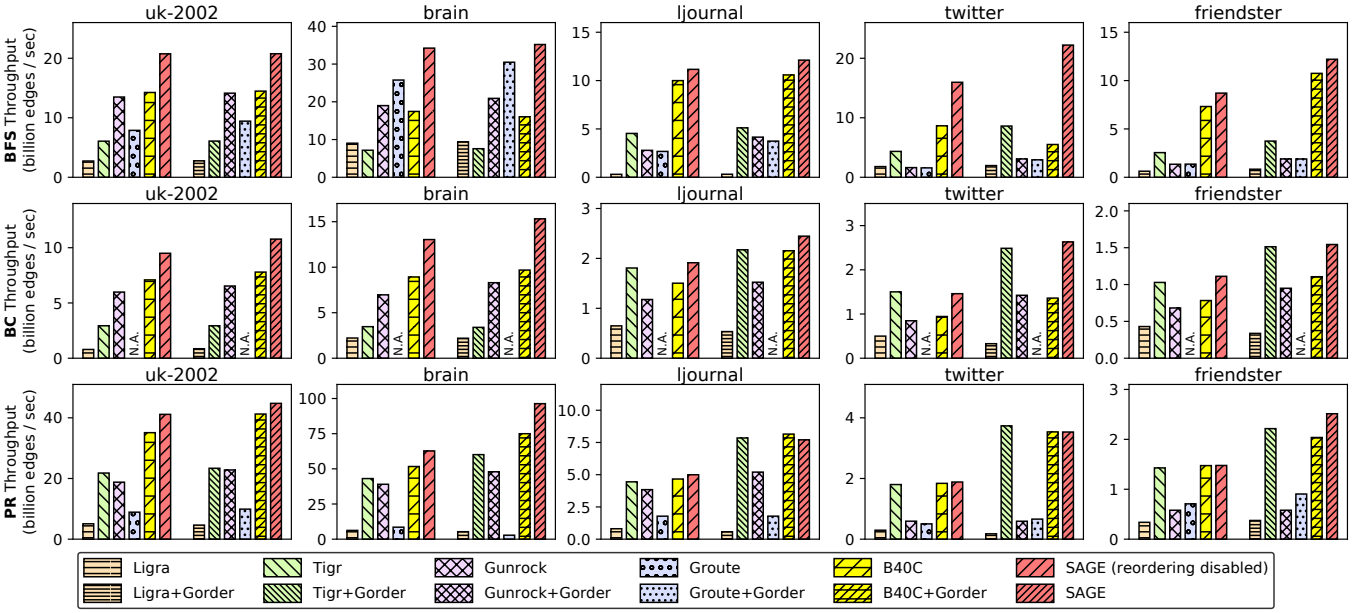

Figure 7: Comparison between SAGE and PGP baselines with/without Gorder.

Table 2: Time Consumption of Reordering (sec.)

Dataset	RCM	LLP	Gorder	SAGE per round
uk-2002	89.2	313.4	45.1	0.1533
brain	33.6	236.7	171.7	0.9932
ljournal	17.4	135.5	103.6	0.0394
twitter	523.5	2737.1	12615.5	1.2894
friendster	654.6	4343.5	15207.7	1.4956

the average. We use the graph traversal speed, i.e., billion edges per second to measure the performance of graph data processing.

Single-GPU Scenario. We evaluate the most common scenario for GPU-accelerated graph analysis, where only one GPU is used and the entire graph data is loaded into the GPU's device memory. We conduct two sets of experiment: **(1) comparison between SAGE and reordering methods; (2) comparison between SAGE and PGP approaches.**

(1) For all compared reordering methods, we generate the corresponding reordered graph replicas and apply SAGE traversal on these replicas to test the influence of different orders on the graph traversal performance. The corresponding experimental results are illustrated in Figure 6. Because the *Sampling-based Reordering* in SAGE is round after round, in Figure 6, the bar with a subscript of **1** denotes the execution based on the original order, and that with a subscript of **100** represents the execution after 99 rounds of reordering the node indices. The sampling threshold is $|E|$, and thus the stage will be switched when the graph representation has responded to the queries of every |E| edges. The time consumption of each reordering method on all evaluated datasets is summarized in Table 2. Through the comparison results, we find that in brain and uk-2002, in most cases, different reordering-based methods do not exhibit much difference from the original order in terms of the graph traversal performance. Nonetheless, in the social network graphs, reordering methods exhibit a relatively remarkable improvement (the largest improvement is in twitter, i.e., up to 36.1%, 80.5%, and 109.3% for the three applications respectively). Among the three evaluated reordering baselines (except SAGE), LLP shows a significant improvement in PR, because LLP is based on local clustering and its optimization goal aligns with the memory access pattern of PR. Nonetheless, Gorder works best in most cases through maxTSP computation, which effectively improves the locality of memory addresses when traversing graph data. However, the preprocessing in Gorder is time-consuming. In contrast, SAGE effectively optimizes the node order for memory efficiency in a lightweight manner, through sampling the tile access. From the experimental results, we can see that the iterative *Sampling-based Reordering* converges fast, i.e., only takes a few rounds to achieve competitive performance compared with Gorder. Particularly, taking twitter in BFS as an example, compared with Gorder, SAGE's *Sampling-based Reordering* can achieve 95.6% graph traversal speed in the 5th round with a negligible cost (i.e., 4.95 seconds); and in the 94th round, SAGE catches up with Gorder, meaning that *Sampling-based Reordering* achieves the same performance at the cost of only 0.95% of Gorder's. Meanwhile, such cost is introduced gradually in the runtime, rather than a large start-up latency caused by preprocessing.

Further, existing reordering methods only assume static graphs, which means that once the graph data is updated, e.g., in dynamic graph analysis scenarios, the effects of preprocessing become invalid and the preprocessing needs to be re-executed. In contrast, SAGE can be directly applied to dynamic graph scenarios as long as the CSR format is used to store the graphs. As our proposed optimizations are fairly lightweight, once the CSR receives new graph updates, we can reorder the graph format quickly by invoking Sampling-based Reordering to speed up subsequent graph processing tasks.

(2) The experimental results between SAGE and PGP approaches are shown in Figure 7. In this comparison, to clarify the interaction between reordering and PGP approaches, we evaluate both scenarios with and without reordering. We apply Gorder to all methods except SAGE, since Gorder performs well for optimizing graph processing performance across different applications and graph datasets as discussed above.

We first analyze the graph processing performance across different approaches and datasets (comparing figures horizontally). According to the comparison between Ligra and other GPU-based methods, we can confirm that GPU-accelerated graph computation can boost the performance by a large margin. Through the comparison results across different datasets, we find that the performance of different methods is related to the types of graph data. Out of the five datasets, for each graph application, it is always the fastest to traverse brain because this graph dataset has a clear hierarchical structure and the distribution of nodes' outdegrees tends to be uniform. In uk-2002, each method also demonstrates a relatively high traversal speed. This is because the web graph is collected by crawlers following hyperlinks and hence, the obtained graph subset has a relatively regular hierarchy as well. Finally, as for the social network graph datasets including ljournal, twitter, and friendster, the traversal speed of each method drops to some extent. The reason is that the distribution of such graph datasets is highly skewed, which poses greater challenges for parallel processing on GPUs. We can clearly find that due to Tigr's UDT preprocessing tailored for irregular graphs, i.e., transforming irregular graphs to a more regular structure through adding auxiliary structures, Tigr has an obvious advantage in terms of performance in social network graphs. However, in brain that is a naturally highly regular graph, Tigr's performance drops severely. This is because Tigr's preprocessing strategy cannot bring more performance improvement when transforming graphs that are already highly regular, e.g., brain. Due to adding auxiliary structures, an excess of overhead is introduced, which degrades the overall performance. Based on the discussion above, the irregularity of graphs is hence a major factor that influences the performance of GPU-based graph processing and is a main concern of further improving the performance of graph processing as well.

For different applications (comparing figures vertically), there are two factors affecting the overall performance, i.e., traversal pattern and atomicity. Among the three applications, BFS and BC are based on local traversal, i.e., the frontiers of each iteration are different subsets of V. In contrast, for PR, it is based on global traversal, i.e., the frontiers of every iteration are the entire V. Hence, applications based on local traversal lead to more irregular workloads, which reflects a more significant performance difference between different approaches. Another performance factor is atomicity. In particular, BFS does not depend on atomic operations when executed in parallel (dirty writes do not affect the correctness of the results); however, BC and PR rely on atomic aggregation. This makes the locality of graphs a double-edged sword that affects performance, i.e., the improved locality leads to better memory load efficiency but increases the conflict of atomic operations.

SAGE exhibits superior performance. In particular, among all the evaluated applications and different kinds of graph datasets,

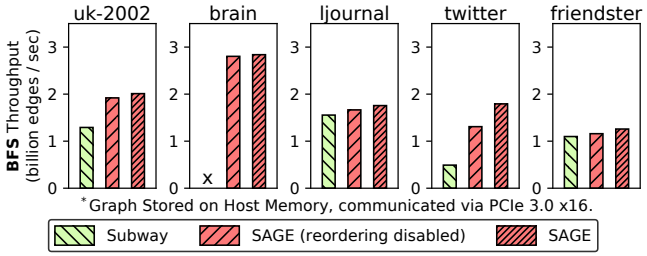

Figure 8: Comparison between SAGE and Subway in the out-of-core scenario.

SAGE consistently achieves the best or a highly competitive performance compared with the best-performing approach. This validates that the proposed SAGE is robust to different workloads and practical for a wide range of applications, whereas none of the baseline methods shows competitive performance in all scenarios. Further, this demonstrates the versatility of SAGE, and confirms that when processing irregular graph processing workloads, *Tiled Partitioning* and *Resident Tile Stealing* in SAGE adapt to the graph data and the hardware without any preprocessing, while fully making use of GPU's computing power in a wide range of situations.

Out-of-core Scenario. Next, we explore the out-of-core scenario where the graph data is not stored in the GPU's device memory and needs to be loaded from the external memory. We evaluate the most common out-of-core scenario, in which the GPU needs to access data from the host memory via PCIe. In this case, the main bottleneck of graph traversal performance lies in the PCIe bandwidth, i.e., the actual performance depends on how effectively the limited external memory bandwidth is utilized. We use Subway as a baseline method in the out-of-core scenario with the comparison results demonstrated in Figure 8[6]. Subway is designed for effectively utilizing the limited PCIe bandwidth. During graph processing, it identifies the active edges in the current computation and preloads the desired subgraph to the local device memory in an asynchronous manner. Such "planned" regular access can contribute to a relatively high actual bandwidth and the asynchronous preloading can decrease the memory access latency. Through the comparison results, we find that SAGE can still achieve satisfactory performance in the out-of-core scenario. The reason is that when SAGE processes irregular graphs, the scattered memory access patterns are avoided due to *Tiled Partitioning*, and the communication efficiency with external memory is further improved due to tile alignment. Meanwhile, *Resident Tile Stealing* can increase the occupancy of the external memory pipeline and therefore, decrease the amortized memory access latency.

Multi-GPU Scenario. Finally, we discuss the multi-GPU scenario in which for a single task, multiple GPUs are used for collaborative processing. The experimental results with multiple GPUs are shown in Figure 9. In the multi-GPU scenario, most existing methods choose to preprocess the graph data, conduct graph partitioning and therefore, achieve a better task dispatching. We use multi-GPU baselines Gunrock and Groute, both of which support the pre-partitioning of graph data based on **metis** [22]. In this way, we compare the two situations (with and without **metis**)

[6]The open-source implementation of Subway will crash in brain.

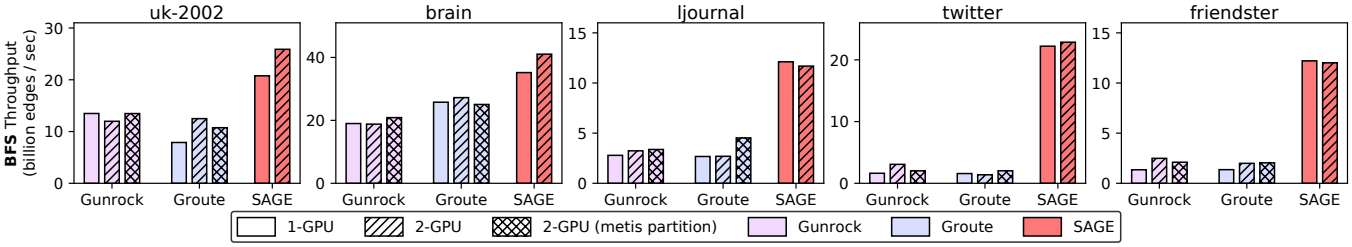

Figure 9: Comparison between SAGE and multi-GPU baselines.

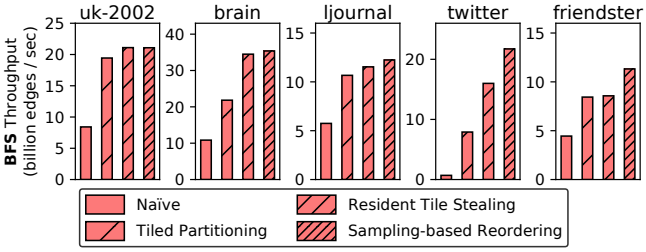

Figure 10: Impact analysis results of the ablation study.

denoted with different patterns in Figure 9. We note that **metis** is time-consuming and its costs are excluded in the reported performance. Further, SAGE is designed without any preprocessing mechanisms. From the experimental results, we can see that all evaluated methods are able to achieve competitive or surpassing performance when using two GPUs with doubled device memory. SAGE achieves the best performance, and particularly in the graph datasets such as `brain` and `uk-2002`.

One can see that using two GPUs does not always lead to better performance, and may perform worse in some cases. However, it can support larger graphs due to the increased size of the device memory. GPU-accelerated graph computation is based on iteratively processing frontier arrays. The computation time of each iteration is short and the data needs to be synchronized after each iteration, which causes the communication overhead between multiple GPUs to be high. The efficient graph analysis with multiple GPUs remains an open and challenging research direction.

7.3 Ablation Study

To validate the impact of our proposed techniques, we apply all these techniques, i.e., *Tiled Partitioning*, *Resident Tile Stealing*, and *Sampling-based Reordering*, incrementally to test their influence on the performance. The experimental results are shown in Figure 10. Meanwhile, Table 3 shows the overhead of Tiled Partitioning, i.e., the percentage out of the total running time, among all datasets and applications. Among different datasets, we have the following observations. To start with, *Tiled Partitioning* exhibits a large impact in all datasets, which again confirms that in GPU-based graph computation, how to effectively handle skewed neighbor distribution to improve the hardware utilization, is a main concern. From the results, we observe that *Tiled Partitioning* exerts a highly remarkable influence on `twitter`. The reason is that although `ljournal` and `friendster` are also social network graphs, they represent private friendship, whereas `twitter` is a public social network, following a popular user does not need a permission. As a result, the skewness of `twitter` is more extreme

Table 3: Tiled Paritioning costs out of running time (msec.)

Dataset	BFS	BC	PR
`uk-2002`	1.6/14.1 (11%)	2.9/28.7 (10%)	0.6/7.1 (8.5%)
`brain`	1.1/16.2 (7%)	2.4/37.4 (6%)	0.1/5.9 (1.7%)
`ljournal`	1.3/7.0 (19%)	2.0/32.9 (6%)	0.2/10.4 (2.0%)
`twitter`	6.7/63.2 (11%)	8.7/540.8 (2%)	2.0/415.7 (0.4%)
`friendster`	15.4/99.3 (16%)	22.5/791.5 (3%)	2.5/718.1 (0.3%)

with the |outdegrees| of some nodes up to several millions. Without proper handling, these super nodes will result in poorer overall performance. *Resident Tile Stealing* flattens the tiles and makes them able to be stolen by all SMs, rather than being processed by only the SM corresponding to the frontier; hence, it raises the parallelism to a large extent. *Resident Tile Stealing* achieves an obvious performance improvement in both `brain` and `twitter` but due to different reasons. As for the former dataset, we think it has a large average |outdegrees|, so *Resident Tile Stealing* can increase the parallelism through flattening tiles in order to achieve lower amortized memory access latency. As for the latter dataset, due to the extremely skewed graph data distribution, *Resident Tile Stealing* solves the inter-SM load imbalance problem and thus, boosts the hardware utilization. The impact of *Sampling-based Reordering* has also been discussed above. In particular, for social network graphs (especially `twitter` and `friendster`), it can achieve a relatively larger improvement, because such data has more potential to raise the locality of accessed memory addresses through assigning a new order to the nodes.

8 CONCLUSION

In this paper, we introduce SAGE, a graph processing framework, specific to high performance and usability of graph analysis on GPUs. First, we propose *Tiled Partitioning* and *Resident Tile Stealing*, to make the graph processing self-adaptive to the hardware and the graph data, without the need for any preprocessing. Second, due to usage of the tile structure, we further optimize the memory efficiency of accessing graph data by *Sampling-based Reordering*. Finally, we evaluate the performance and applicability of the proposed SAGE through extensive experiments in representative graphs and application scenarios for three graph applications.

9 ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments. This project is partially supported by a MoE Tier 2 grant (MOE2017-T2-1-141) in Singapore. Yuchen's work is in part supported by a MoE Tier 2 grant (MOE2019-T2-2-065) in Singapore.

REFERENCES

[1] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In *PPOPP*. ACM, 105–114.

[2] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. 2011. Computing Strongly Connected Components in Parallel on CUDA. In *IPDPS*. IEEE, 544–555.

[3] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *PPOPP*. ACM, 235–248.

[4] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: a scalable fully distributed Web crawler. *Softw. Pract. Exp.* 34, 8 (2004), 711–726.

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *WWW*. ACM, 587–596.

[6] Gregory Buehrer and Kumar Chellapilla. 2008. A scalable pattern mining approach to web graph compression with communities. In *WSDM*. ACM, 95–106.

[7] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *KDD*. ACM, 219–228.

[8] Elizabeth Cuthill and James McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*. 157–172.

[9] Nhat Tan Duong, Quang Anh Pham Nguyen, Anh Tu Nguyen, and Huu-Duc Nguyen. 2012. Parallel PageRank computation using GPUs. In *SoICT*. ACM, 223–230.

[10] Alan George and Joseph W Liu. 1981. *Computer solution of large sparse positive definite*. Prentice Hall Professional Technical Reference.

[11] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David A. Bader. 2020. Traversing Large Graphs on GPUs with Unified Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1119–1133.

[12] Laurence Goasduff. 2020. *Top 10 trends in data and analytics for 2020.* Retrieved Sep 1, 2020 from https://www.gartner.com/smarterwithgartner/gartner-top-10-trends-in-data-and-analytics-for-2020/

[13] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD Conference*. ACM, 1067–1082.

[14] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel Personalized Pagerank on Dynamic Graphs. *Proc. VLDB Endow.* 11, 1 (2017), 93–106.

[15] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting Reuse for GPU Subgraph Enumeration. *IEEE Trans. Knowl. Data Eng.* (2020).

[16] Mark Harris and Kyrylo Perelygin. 2017. *Cooperative Groups: Flexible CUDA Thread Programming*. Retrieved Sep 1, 2020 from https://developer.nvidia.com/blog/cooperative-groups/

[17] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. 2017. Fast segmented sort on GPUs. In *ICS*. ACM, 12:1–12:10.

[18] Jayadharini Jaiganesh and Martin Burtscher. 2018. A high-performance connected components implementation for GPUs. In *HPDC*. ACM, 92–104.

[19] Zhihao Jia, Yongkee Kwon, Galen M. Shipman, Patrick S. McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *Proc. VLDB Endow.* 11, 3 (2017), 297–310.

[20] DS Johnson and L Stockmeyer. 1976. Some simplified NP-complete graph problems. *Theoretical Computer Science* 1 (1976), 237–267.

[21] Chinmay Karande, Kumar Chellapilla, and Reid Andersen. 2009. Speeding up algorithms on compressed web graphs. In *WSDM*. ACM, 272–281.

[22] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[23] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *HPDC*. ACM, 239–252.

[24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. 2010. What is Twitter, a social network or a news media?. In *WWW*. ACM, 591–600.

[25] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin C. Herbordt. 2014. An investigation of Unified Memory Access performance in CUDA. In *HPEC*. IEEE, 1–6.

[26] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiaoli Li. 2017. Network Motif Discovery: A GPU Approach. *IEEE Trans. Knowl. Data Eng.* 29, 3 (2017), 513–528.

[27] Hang Liu, H. Howie Huang, and Yang Hu. 2016. iBFS: Concurrent Breadth-First Search on GPUs. In *SIGMOD Conference*. ACM, 403–416.

[28] Shengliang Lu, Bingsheng He, Yuchen Li, and Hao Fu. 2021. Accelerating exact constrained shortest paths on GPUs. *Proceedings of the VLDB Endowment* 14, 4 (2021), 547–559.

[29] Enrico Mastrostefano and Massimo Bernaschi. 2013. Efficient breadth first search on multi-GPU systems. *J. Parallel Distributed Comput.* 73, 9 (2013), 1292–1305.

[30] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.* 1, 2 (2015), 14:1–14:30.

[31] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. 2021. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. *Proc. VLDB Endow.* 14, 2 (2021), 114–127.

[32] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *IPDPS*. IEEE Computer Society, 463–474.

[33] Diego Piccinotti, Edoardo Ramalli, Alberto Parravicini, Rolando Brondolin, and Marco D. Santambrogio. 2019. Solving write conflicts in GPU-accelerated graph computation: A PageRank case-study. In *RTSI*. IEEE, 144–148.

[34] Gerhard Reinelt and Hanna Seitz. 2014. On a binary distance model for the minimum linear arrangement problem. *Top* 22, 1 (2014), 384–396.

[35] Arnon Rungsawang and Bundit Manaskasemsak. 2012. Fast PageRank Computation on a GPU Cluster. In *PDP*. IEEE, 450–456.

[36] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[37] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *ASPLOS*. ACM, 622–636.

[38] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *EuroSys*. ACM, 12:1–12:16.

[39] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2017), 420–431.

[40] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (2017), 107–120.

[41] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2019. GPU-based Graph Traversal on Compressed Graphs. In *SIGMOD Conference*. ACM, 775–792.

[42] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP*. ACM, 135–146.

[43] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *IPDPS Workshops*. IEEE, 1–8.

[44] John A Stratton, Nasser Anssari, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Daniel Liu, and Wen-mei Hwu. 2012. Optimization and architecture effects on GPU computing workload performance. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–10.

[45] William F Tinney and John W Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809.

[46] Ha Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *DASFAA (1) (Lecture Notes in Computer Science, Vol. 9049)*. Springer, 299–315.

[47] Koji Ueno and Toyotaro Suzumura. 2013. Parallel distributed breadth first search on GPU. In *HiPC*. IEEE Computer Society, 314–323.

[48] Yangzihao Wang, Yuechao Pan, Andrew A. Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1 (2017), 3:1–3:49.

[49] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *SIGMOD Conference*. ACM, 1813–1828.

[50] Hao Wen and Wei Zhang. 2019. Improving Parallelism of Breadth First Search (BFS) Algorithm for Accelerated Performance on GPUs. In *HPEC*. IEEE, 1–7.

[51] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *SC*. IEEE / ACM, 60:1–60:13.

[52] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *ICDM*. IEEE Computer Society, 745–754.

[53] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1543–1552.