CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor (Technical Report)

Pengjie Cui[1], Haotian Liu[2], Bo Tang[2], Ye Yuan[3]
[1] Northeastern University, [2] Southern University of Science and Technology, [3] Beijing Institute of Technology
[1] 1810602@stu.neu.edu.cn [2] {12231144@mail, tangb3@}.sustech.edu.cn [3] yuan-ye@bit.edu.cn

Abstract

In recent years, numerous CPU-GPU heterogeneous graph processing systems have been developed in both academic and industrial to facilitate efficient large-scale graph processing in various applications, including social networks and biological networks. However, the performance of existing systems can be significantly enhanced due to the presence of two outstanding challenges: GPU memory over-subscription and efficient CPU-GPU cooperative processing.

In this work, we propose CGgraph, an ultra-fast graph processing system on modern commodity CPU-GPU co-processor. In particular, CGgraph overcomes GPU-memory over-subscription by extracting a subgraph which only need to load into GPU memory once, but its vertices and edges can be used in multiple iterations during the graph processing procedure. Specifically, we devise a graph reordering algorithm and propose a subgraph extraction approach to achieve that. To support efficient CPU-GPU co-processing, we design a CPU-GPU cooperative processing scheme, which balances the workloads between CPU and GPU by on-demand task allocation. Moreover, a suite of optimization techniques (e.g., vertex-level and edge-level task stealing, intra-warp balancing) have been proposed to balance the workloads on CPU and GPU.

To evaluate the efficiency of CGgraph, we conduct extensive experiments to compare it with 7 state-of-the-art systems by running 4 well-known graph algorithms on 6 real world graphs. Our prototype system CGgraph outperforms all existing systems, delivering up to an order of magnitude improvement. Moreover, CGgraph on a modern commodity machine with CPU-GPU co-processor yields superior (or at the very least, comparable) performance compared to existing systems on a high-end CPU-GPU server.

1 Introduction

Graph data have been widely used in various domains, e.g., social networks [12], web content [50], and biological networks [36]. The scale of graphs often undergoes significant growth over time in various graph applications. The *vertex-centric computation model*, as introduced in the Pregel [32], has become the *de facto* standard for efficient graph processing in many graph applications due to its simplicity and scalability. This model has been widely adopted by various CPU-based distributed systems [19, 20, 27, 32, 35, 42, 53, 55] for processing large-scale graphs. In recent years, to enhance the performance of large-scale graph processing, both academic and industrial communities have developed many GPU-based systems [9, 18, 24, 30, 33, 38, 41, 46, 48, 52, 54] by leveraging the substantial parallelism offered by general-purpose graph processing units (GPUs). The existing GPU-based systems can be categorized into two groups: (i) GPU-only computing systems, such as Cusha [24], Gunrock [48], Groute [9]; and (ii) CPU-GPU heterogeneous computing systems, including Totem [18], Subway [38], LargeGraph [52]. While GPU-only systems offer high performance, they face limitations in processing large graphs that exceed the GPU global memory capacity, also known as *GPU memory over-subscription*. To address it, CPU-GPU heterogeneous systems have been developed to mitigate the issue by leveraging both CPU and GPU resources. However, despite these advancements, there are still two opening challenges that need to be addressed in order to build an ultra-fast graph processing system on a modern commodity CPU-GPU co-processor.

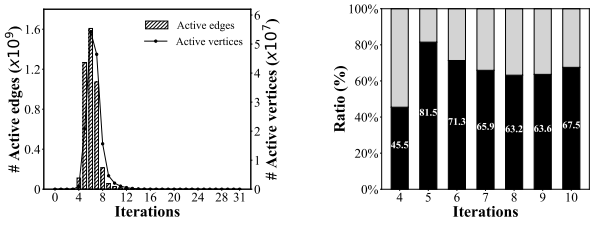
GPU memory over-subscription. Out-of-core graph processing has emerged as a prevalent approach to tackle GPU memory over-subscription in CPU-GPU heterogeneous systems. In particular, three prominent approaches in the realm of out-of-core graph processing have been explored in the literature.

(I) Partition-based approach [21, 25, 40]: The large graph is partitioned, then loaded "partitions" to the limited GPU global memory before each iteration during the graph processing procedure. While this approach has been improved by various techniques, such as asynchronously streaming data movement [21, 25, 40], and reducing data movement size by tracking active vertices or edges [21, 40], the benefits are still limited. This is primarily because the data movement cost in each iteration remains expensive due to the constrained bandwidth of the PCIe connection between the CPU's main memory and the GPU's global memory.

(II) Unified memory-based approach [17, 29]: With the advent of CUDA 8.0 and the Pascal architecture, GPU applications gained the ability to transparently access the CPU main memory. Thus, many graph processing system loads data pages from CPU main memory to GPU global memory by triggering page faults. This approach offers the advantage of easy programming and on-demand data loading. However, this approach still faces two major challenges that affect its performance: (i) the page faulting process incurs overhead; and (ii) the on-demand loading of pages may include a large proportion of inactive edges/vertices, which results to unnecessary data transfers.

(III) Subgraph generation-based approach [38]: Recently, a subgraph generation algorithm is proposed in Subway [38] to tackle GPU memory subscription, which generates a subgraph that only includes active edges and vertices in GPU memory. A suite of techniques (e.g., concise-and-efficient graph representation, GPU accelerated implementation) have been devised to optimize its generation cost. This approach has proven to be effective in improving performance by reducing the frequency and the size of loaded subgraph. Nevertheless, there is room for improvement as the generation of the subgraph in nearly every iteration incurs costs.

Efficient CPU-GPU Cooperative Processing. Existing CPU-GPU heterogeneous graph processing systems leverage both CPU and GPU to execute graph algorithms. However, the methods of



(a) The number of active edges and vertices of each iteration

(b) The ratio of active edges in GPU global memory of CGgraph

Figure 1: The statistics of running SSSP on *friendster*

cooperation between the CPU and GPU differ significantly across these systems, leading to variations in graph processing performance. We next summarize the CPU-GPU cooperative methods.

(I) CPU and GPU are the first-class citizens [18]: Both CPU and GPU execute the computation tasks in each iteration during graph processing procedure in this method. For example, Totem [18] first loads the partition to GPU memory at the beginning, then the GPU cores will process the tasks of the active edges and vertices in its memory, and the rest active edges and vertices will be executed by CPU cores in each iteration. The CPU and GPU are executing different computation tasks in each iteration. Nonetheless, the workload distribution between the CPU and GPU can vary significantly across iterations due to the partitioning of the graph load on the GPU.

(II) GPU is the first-class citizen [21, 38, 40]: In this method, the GPU is primarily responsible for executing the computational tasks in each iteration. The CPU plays a supportive role and assists the GPU in various ways. For instance, the CPU identifies the active vertices that serve as input for the GPU-accelerated subgraph generation algorithm in Subway [38]. There is a clear opportunity for improving the utilization of the CPU since it may be idle while the GPU is heavily occupied with task processing.

(III) Hybrid roles of CPU [52]: This method combines the utilization of the CPU in both (I) and (II). The CPU initially acts as an assistant to the GPU, as described in (II). However, it also executes partial computation tasks during each iteration of graph processing. For example, in LargeGraph[52], the CPU first identifies the frequent paths in the graph and assign the tasks which are in frequent paths to GPU. After that, it also processes the computations task in infrequent path. Nevertheless, the method still encounters task-level imbalances during graph processing. This is primarily due to the significant difference in the number of active edges for a vertex in the frequent path compared to its counterpart in the infrequent path.

In this work, we propose an ultra-fast graph processing system CGgraph, which simultaneously addresses the above two open challenges, i.e., GPU memory over-subscription and efficient CPU-GPU cooperative processing. Moreover, its performance improvement does not rely on the high-end CPU-GPU servers as it is built on a modern commodity machine with CPU-GPU co-processor.

As the above discussion, the key limitation of all existing solutions for GPU memory over-subscription is the benefits obtained by exploiting high parallelism of GPU is neglected by the data movement overhead. In CGgraph, we address it by following *"load once but use multiple times"* principle. Specifically, Figure 1(a) depicts the number of active edges and vertices in each iteration by running

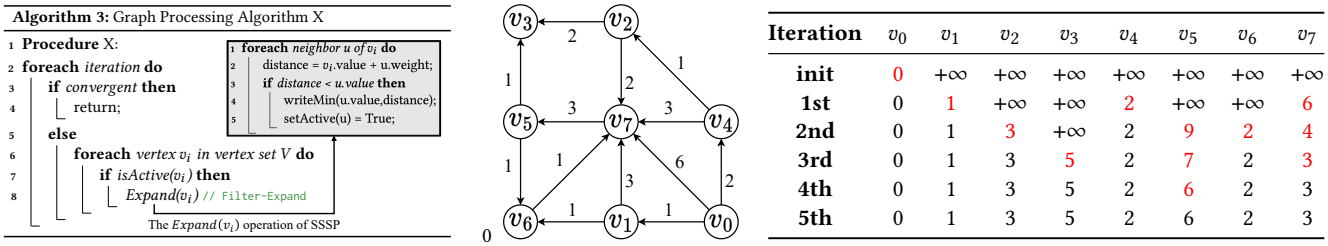
Table 1: The range of speedup times of CGgraph

Dataset	BFS	SSSP	WCC	PR
gsh2015tpd	0.74-2.35X	1.22-4.44X	0.97-4.72X	1.25-2.23X
twitter2010	1.53-3.39X	2.18-4.68X	2.01-5.11X	2.70-**9.73X**
friendster	2.05-3.48X	4.45-**11.94X**	3.42-**11.07X**	2.66-7.83X
weibo	2.11-2.94X	2.43-6.05X	2.81-7.50X	2.39-6.35X
uk-2006	0.96-**4.15X**	2.46-5.37X	1.58-3.91X	2.33-6.96X
un-union	1.30-2.86X	2.05-2.61X	1.45-2.82X	1.79-2.79X

single source shortest path (SSSP) algorithm on *friendster*, which includes 124 millions of vertices and 1,806 millions of edges. It is evident that the number of active edges and vertices undergoes significant variation across iterations. Hence, the general idea to address GPU memory over-subscription in CGgraph is extracting a size-constrained subgraph G' from graph G, in which its vertices and edges can be active in multiple iterations. Thus, CGgraph loads it to GPU global memory before graph processing and keeps it in GPU during the whole graph processing procedure. To achieve that, we propose two key techniques: (i) graph reordering, and (ii) size-constrained subgraph extraction, in CGgraph. Figure 1(b) reports the ratio of active edges (ranges from 45.5% to 81.5%) in CGgraph's extracted subgraph (loaded into GPU memory) over 4th to 10th iterations by running SSSP on *friendster*, which confirms the effectiveness of our solution for GPU memory over-subscription in CGgraph. Interestingly, our proposed graph reordering approach is generic which can be directly used in existing graph processing systems to improve their performance, as we will elaborate shortly.

For the CPU-GPU cooperative processing, the first pitfall of existing solutions is that the GPU should be used in every iteration during graph processing. As shown in Figure 1(a), the overhead to invoke GPU definitely will be larger than its benefits at those iterations which only has few active edges and vertices, e.g., 1st-3rd iterations and 11th to 31st iterations. In CGgraph, we devise a simple-yet-effective GPU invoking strategy to decide whether the GPU will be invoked or not in each iteration. The second and the most important limitation of existing solutions is that almost all of them (if not all) lack of fine-grained (i.e., vertex-level and edge-level) CPU and GPU workload balance scheme. In CGgraph, we devise (i) an edge-level CPU-GPU cooperative computation scheme and (ii) on-demand task allocation approach to guarantee fine-grained workload balance between CPU and GPU. To the best of our knowledge, CGgraph is the first graph processing system which offers cooperative CPU and GPU processing on both edge-level and vertex-level. To further unlock the computation capabilities of CPU and GPU, a suite of optimizations (e.g., task stealing, inter-warp balancing strategy) for task processing on CPU and GPU have been equipped in CGgraph.

We conduct extensive experiments to demonstrate the superiority of our proposed CGgraph with 7 state-of-the-art systems. Specifically, we report the minimum and maximum speedup times of CGgraph over all these 7 competitor systems by running 4 widely used graph algorithms, i.e., Breadth-First-Search (BFS), Single-Source Shortest Path (SSSP), Weakly Connected Components (WCC), and PageRank (PR) on 6 public graph datasets in Table 1. CGgraph achieves up to 4.15X, 11.94X, 11.07X and 9.73X speedup over existing systems by processing BFS, SSSP, WCC and PR, respectively. To our surprise, on a modern commodity machine equipped with a



(a) Vertex-centric computation model (b) Graph G

Iteration	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
init	0	+∞	+∞	+∞	+∞	+∞	+∞	+∞
1st	0	1	+∞	+∞	2	+∞	+∞	6
2nd	0	1	3	+∞	2	9	2	4
3rd	0	1	3	5	2	7	2	3
4th	0	1	3	5	2	6	2	3
5th	0	1	3	5	2	6	2	3

(c) Iterations of SSSP algorithm

Figure 2: Graph computation model and its illustrative example of SSSP algorithm

CPU-GPU co-processor, CGgraph exhibits superior performance compared to existing systems running on high-end CPU-GPU servers. This remarkable achievement highlights the efficiency and capability of CGgraph in leveraging CPU and GPU hardware resources effectively, surpassing the performance of established systems.

The rest of the paper is organized as follows. In Section 2, we present graph processing computation model and its research challenges. Section 3 overviews the architecture of CGgraph. The techniques to address GPU memory over-subscription problem and efficient CPU-GPU cooperative computation problem are introduced in Sections 4 and 5, respectively. The experimental evaluation results are presented in Section 6. We conclude the paper and highlight the future research direction in Section 7.

2 Computation Model and Its Challenges

In this section, we first present the widely-used computation model in graph processing systems in Section 2.1, then we highlight the research challenges to build a fast graph processing system on a modern commodity machine with CPU-GPU co-processor in Section 2.2.

2.1 Graph Processing Computation Model

Many graph processing systems adopt a vertex-centric computation model (a.k.a., programming model) since its high scalability [19, 48, 55]. As Figure 2(a) depicted, the vertex-centric computation model employs an "iterative-convergent" process. In particular, the given graph algorithm is processed by iteration-manner, and it terminates when all vertex values do not change or the procedure reaches a limited iteration time, known as "convergent" in Line 3, Algorithm 1. For each iteration, every vertex in the graph is processed by *Filter-Expand* operation [33, 34]. We refer to the *Filter-Expand* operation of a vertex as a computation *task* in the graph processing system. In particular, the *Filter* step selects the active vertices to execute the *Expand* (see Line 7), and the inactive vertices are filtered. The *Expand* step processes the input active vertex, and generates new active vertices for the next iteration. More specifically, the *Expand* step maintains algorithm-specific logic for vertex processing, which ensures that all active vertices are processed with the same algorithmic logic. The inside box in Figure 2(a) shows an example of the *Expand* step for SSSP algorithm, which computes the shortest distance from a given source vertex to all other vertices on a weighted directed graph. The vertex-centric computation model is generic as it can be instantiated to various graph processing algorithms, e.g., BFS, PR, WCC, by only changing the algorithmic logic in the above *Expand* step. Moreover, the computation tasks of all vertices in each iteration can be parallel processed inherently via high-parallelism hardware (e.g., multi-threads CPU or GPU).

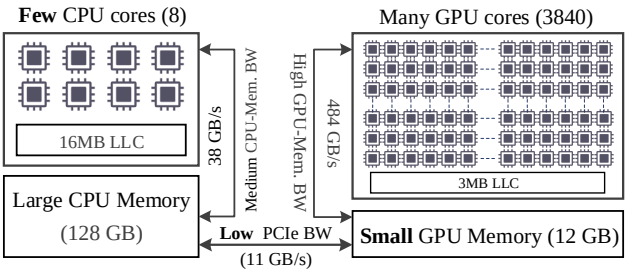


Figure 3: The architecture of CPU-GPU co-processor

Example. Taking the graph G in Figure 2(b) as an example, we illustrate the vertex-centric computation processing procedure of SSSP with source node v_0. At the beginning iteration, the vertex values, i.e., the found shortest distance from the source vertex v_0 so far, are initialized to +∞. For *Expand* step of v_0, it updates its vertex value to 0 and active its neighbors $\{v_1, v_4, v_7\}$, which will be processed in the next iteration. In the 1st iteration, the vertex values of v_1, v_4 and v_7 are computed by its corresponding *Expand* step, see the red values in the 1st iteration row, Figure 2(c). The newly active vertex set is $\{v_2, v_5, v_6, v_7\}$, which will be processed in the 2nd iteration. As illustrated in Figure 2(c), the processing procedure terminates after the 5th iteration as all the shortest distances from source v_0 are computed.

In the literature, subgraph-centric computation model is also used in various graph processing systems [37, 43, 44]. However, it is not trivial to accelerate it by CPU-GPU co-processing [22], e.g., how to process and synchronize among subgraphs? How to exploit single instruction multiple threads architecture of GPU to parallel process different subgraphs? We left it as future work as it is out-the-scope of this work.

2.2 Research Challenges

Following the widely used vertex-centric computation model, we focus on building an ultra-fast graph processing system on a single modern commodity machine via CPU-GPU co-processing in this work. In this section, we first introduce the characteristics of the modern commodity machine with CPU-GPU co-processors, then summarize the research challenges to build an ultra-fast graph processing system upon it.

CPU-GPU Co-processor. Figure 3 shows the architecture of a CPU-GPU co-processor in a modern commodity machine, i.e., DELL OptiPlex 7000MT Desktop Computer. There are several important

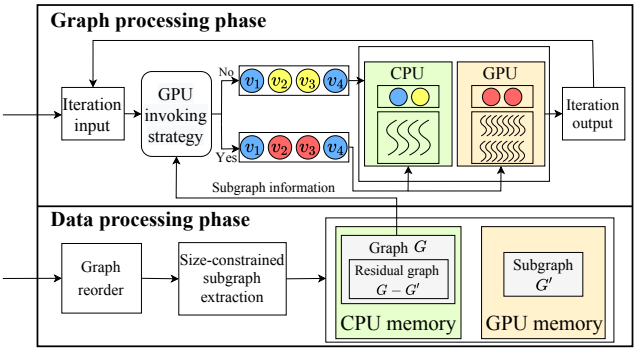


Figure 4: CGgraph architecture overview

properties in a CPU-GPU co-processor. First, the GPU has a larger number of cores (i.e., 3840 cores) when compares with the CPU (with 8 cores), which enables high parallelism computation in GPU. Second, GPU utilizes high-bandwidth memory (HBM), which is higher (484 GB/s) than the memory access bandwidth of the CPU (38 GB/s). Last but the most important, the size of the CPU main memory is always significantly larger than the size of the GPU global memory, e.g., 128GB vs. 12GB. The data movement from CPU memory to GPU memory is very expensive as the bandwidth of PCIe is limited, e.g., 11 GB/s, which is the core challenge to design an out-of-GPU-memory data processing system [21, 31, 38, 40, 52].

Research Challenges. Many CPU-GPU heterogeneous graph processing systems incur poor performance as: (i) do not fully utilize the computation power of CPU and GPU; and (ii) the parallel computing benefit gained from GPU is far smaller than the extra data movement cost. In this work, we want to build an ultra-fast graph processing system, which fully utilizes the computing units of the CPU-GPU co-processor in a single modern commodity machine. However, there are two research challenges to achieve that:

- C1: how to utilize the GPU memory efficiently? As elaborated in Section 1, existing systems utilize GPU global memory by (i) loading data partitions into it at each iteration explicitly [21, 40, 52], and (ii) generating subgraph which only includes active edges and vertices at nearly every iteration [38]. However, both methods incur significant overheads, e.g., the data movement cost or subgraph generation cost. Thus, the first challenge of our work is utilizing the limited GPU video memory efficiently.
- C2: how to cooperate CPU and GPU effectively? Specifically, existing systems either use CPU memory as the secondary storage [21, 38, 40] or assign computation tasks to CPU via a fixed strategy [18, 52]. The limitations are two-fold: (i) the co-processing between CPU and GPU is implicit, which results in workload imbalance, and (ii) they do not fully unlock the computation capabilities of CPU and GPU. Hence, the second challenge of our work is co-operating CPU and GPU effectively?

3 The Overview of CGgraph

Figure 4 depicts the architecture of our proposed CGgraph, it consists of two phases: (i) data processing phase, and (ii) graph processing phase, which are designed to address the two challenges mentioned before in Section 2, respectively. In particular, the raw graph data is pre-processed in the data processing phase. Then CGgraph extracts a size-constrained subgraph G' from graph G, in which its vertices and edges can be active in multiple iterations, and loads to GPU global memory. We will elaborate on the technical details and their benefits in Section 4. For the graph processing phase, CGgraph employs the widely used vertex-centric computation model as other studies (see Section 2) to process graph algorithms. For each iteration, the GPU invoking strategy first decides how to process the active vertices/edges, i.e., CPU-only or CPU-GPU co-processing by exploiting the statistics of the subgraph G'. Then, all computation tasks will be executed accordingly. The details of the proposed techniques in the graph processing phase are described in Section 5.

4 Data Processing Phase in CGgraph

As the data movement cost between CPU main memory and GPU video memory is very expensive, the principle of our solution is: "load once and use multiple times!" With the above idea in mind, our research goal is extracting a subgraph G' from the input large graph G, such that (i) only load it to GPU global memory once, and (ii) it unlocks the high parallelism computation capability of GPU among multiple iterations. In this section, we first analyze the hardness of subgraph G' extraction problem in Section 4.1. We then highlight our observed insights in the graph processing computation model (see Section 4.2), which guide us to design the two core subroutines in our solution: graph reordering in Section 4.3, and size-constrained subgraph extraction algorithm in Section 4.4.

4.1 Hardness Analysis

We first define it formally at Problem 1 as follows.

Problem 1. *Given the input graph $G(V, E)$ and GPU memory size M, the subgraph extraction problem (SEP) is extracting a subgraph $G'(V', E')$ from G such that the size of G equals to M.*

During the subgraph extracting process, for a node $v_i \in V'$, its all out-edges in G should also be included in E' of G' to improve the GPU memory access latency and adapt the vertex-centric computation model. We then analyze the hardness of the subgraph G' extraction problem by Lemma 1.

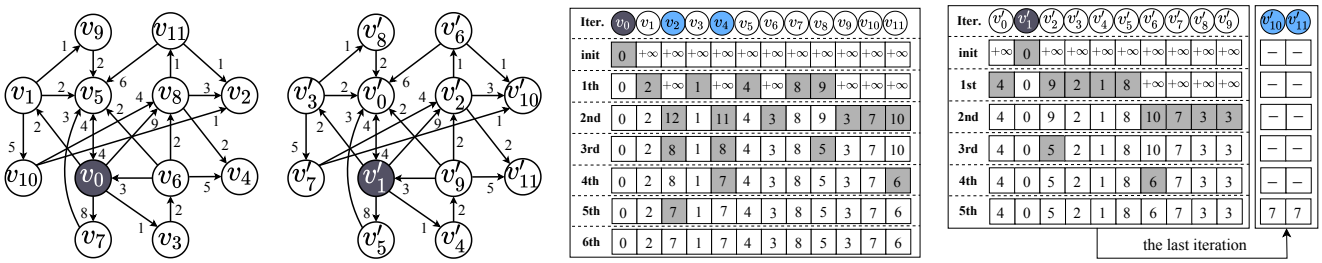
Lemma 1. *The subgraph extraction problem is NP-hard.*

Proof. The proof idea is transforming the subset sum problem (SSP) [26], which is a well-known NP-hard problem, to our subgraph extraction problem (SEP). We omit the proof details due to space limitations and refer the interested reader to our technical report [1]. □

4.2 Observed Insights

Due to the hardness of the subgraph extraction problem, see Problem 1, in this section, we present two interesting insights into the vertex-centric graph processing computation model, which sheds light on how to extract subgraph G' from graph G heuristically-yet-effectively.

Returning back the SSSP example (in Figure 2) of the graph processing computation model in Section 2, we found vertex values of v_5 and v_7 are updated in three iterations during the graph processing procedure. More specifically, the vertex value of v_7 is updated



(a) The original graph (b)The graph after reordering (c) SSSP on the original graph (d) SSSP on the reordered graph

Figure 5: The effect of graph reordering on SSSP algorithm

by $v_0 \rightarrow v_7$, $v_0 \rightarrow v_1 \rightarrow v_7$, and $v_0 \rightarrow v_1 \rightarrow v_6 \rightarrow v_7$ in the 1st, 2nd, and 3rd iteration, respectively. Moreover, the vertex value v_5 also be updated in the 2nd to 4th iterations as it is an out-going neighbor of v_7. We then summarize the first insight according to the above observation.

Insight 1. The number of active times of different vertices in the graph are quite different. These vertices which have a large number of in-coming edges are probably will be active in many iterations during the query processing procedure. Consequently, the out-going neighbor of such vertices also will be affected in many iterations.

Another interesting fact is that v_3 is a special type of vertices (a.k.a., sink vertices), which do not have any out-going neighbors, thus, the vertex value of v_3 does not influence any other vertex value in the whole processing procedure. Moreover, the vertex value of v_3 depends on the vertex values of is in-coming neighbors v_2 and v_5. Hence, the second insight we observed is as follows.

Insight 2. The sink vertices do not influence other vertices, thus, their vertex values will not affect the value of other vertices. Specifically, the vertex value of a sink vertex can be computed after all its in-coming neighbors' vertex values are finalized, for example, compute them at the last iteration.

4.3 Graph Reordering

With the above observed insights, a general idea is to selectively load the vertices that have a high number of incoming neighbors to the GPU's constrained memory, while excluding the sink vertices. This problem is a variant of the well-known dense subgraph extraction problem [28]: *given a sparse graph, how to find a meaningful dense subgraph?* One of the commonly used methods is reordering and positioning the vertices or edges to a specific location, then extracting a size-constrained subgraph upon the reordered graph. For example, Chen et, al. [11] reorders the adjacency matrix and gathers the dense part into the diagonal, then proposes a blocking algorithm to find the dense diagonal block. Inspired by it, we devise a graph reordering algorithm which stores the vertices with more in-coming neighbors at the front while the sink vertices at the end. In addition, we observed that when a vertex finishes its computation, its neighbors are likely to be active in the next iteration during the graph processing. We explicitly take this observation into consideration to enhance data access locality. Therefore, the reordering algorithm ensures that the out-going neighbors of a vertex with large in-coming neighbors are situated nearby.

To achieve the above goals, we propose a heuristic graph reordering algorithm as follows, which takes all vertices V of the graph as input, and outputs the reordered list S'_V. It sorts all vertices in V by the descending order of its in-degree at first, denoted by S_V. Then we slightly revise the Breath-First-Search (BFS) algorithm to traverse all these vertices in the graph to obtain the reordered list S'_V. Specifically, the revised BFS algorithm starts with the first non-visited vertex in the sorted vertex array S_V. During the traversal procedure, each visiting vertex is either appended to the reordered list S'_V (i.e., non-sink vertex) or appending to the sink vertices list S_{sink} (i.e., sink vertex). The above traversal steps repeat until all vertices have been visited. Last, the sink vertices list S_{sink} is appended to the reordered list S'_V.

We next demonstrate the effectiveness of placing the sink vertices at the end and the extra benefits of the improved data access locality via the above graph reordering algorithm. We left how to extract the size-constrained subgraph in Section 4.4.

Example. Figure 5(a) shows the original graph, and its reordered graph is presented in Figure 5(b). The subscripts of v and v' represent the order of a vertex in the original graph and reordered graph, respectively. The graph reordering algorithm changes the id of each vertex, e.g., v_0 in the original graph is the same as v'_1 in the reordered graph. We run the SSSP algorithm on both graphs. Their corresponding iterations are shown in Figures 5(c) and (d), respectively. In particular, the sink vertices in the reordered graph are located at the end, which are marked as blue. We update them only after the vertex values of other vertices have been finalized. Obviously, moving the sink vertices to the end and postponing its vertex value computation via the reordered graph significantly reduces the computation cost when we compare the iterations in the original graph and reordered graph. For example, the vertex values of sink vertices v_2 and v_4 are updated three times, see the columns with blue color vertex in Figure 5(c). However, they are only updated once in the reordered graph, as the last two columns in Figure 5(d).

Moreover, the graph reordering algorithm explicitly improves data access locality in each iteration. In both Figures 5(c) and (d), the gray cells in each row are the accessed vertices in each iteration. Visually, the data access locality on reordered graph is preserved much better than the original graph, as the visited vertices of each iteration in Figure 5(d) are located in continuous grids. Taking the 2nd iteration as an example, the original graph visits v_2, v_4, v_6, v_9, v_{10}, and v_{11}. However, four of these vertices are consecutively

positioned from v'_6 to v'_9 in the reordered graph, and two are sink vertices that do not need access at this iteration.

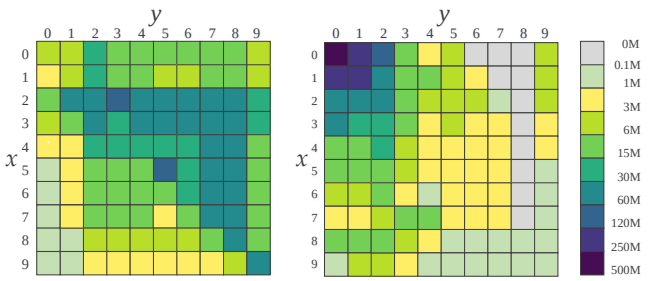
Relevant studies discussion. Many graph reordering algorithms [10, 13, 14, 16, 49, 51] have been studied in the literature for various optimization goals. For example, RCM [13] rearranges the matrix representation of a graph into a band matrix with a narrow bandwidth. Gorder [49] enhances the average neighborhood overlap of connections between adjacent nodes by maximizing the shared edges within blocks of consecutive nodes (sliding window) with size w to reduce the cache miss. Adapting them to our problem are possible, but their performance will be worse than our above graph reordering algorithm as the goal of us is reordering the vertices by their in-degree and explicitly considering the data access locality, we will verify it by experiments in Section 6.

Generality discussion. It is worth to point out the above graph reordering algorithm can be used to improve the performance of all existing graph processing systems (e.g., Totem [18], Subway [38]) as it enhances the data access locality in an explicit way, we will evaluate its effectiveness on existing systems in Section 6.

4.4 Size-constrained Subgraph Extraction

Before we introduce our size-constrained subgraph extraction approach, we first demonstrate the effectiveness of our graph reordering algorithm, which moves the vertices with high in-coming neighbors to the front. Figures 6(a) and (b) shows the heatmap of edge distributions among the vertex group pairs of *twitter2010* in the original and reordered graph, respectively. The deeper (lighter) the color of a cell, the larger (smaller, respectively) the edges in this vertex group pair. Specifically, *twitter2010* has 42 millions of vertices, which are equally divided into 10 groups. For example, the first 4.2 millions vertex are assigned to group 0. An edge (v_i, v_j) in the graph G is counted in vertex group pair (x, y) if and only if its start vertex v_i is in the vertex group x and its end vertex v_j is in the vertex group y. Comparing with the heatmap of the original graph in Figure 6(a), the deeper color cells are concentrated on the left-upper corner of the reordered graph heatmap in Figure 6(b). The reason is that our graph reordering algorithm moves the vertices with high in-coming neighbors to the front, thus, the group pairs at the left-upper corner are probably include more edges than others. In addition, it is no surprise that the color of the cells in right-bottom corner is very light as these are sink vertices or vertices with few in-coming neighbors.

With the reordered graph, we next present our size-constrained subgraph extraction method as follows. The general idea is that the extracted subgraph should include more vertices in the left-upper corner in Figure 6(b). The steps of size-constrained subgraph extraction method are: First, it extracts a subset of vertices from the beginning of the ordered vertex list. The way to determine the number of the vertices to be extracted with the limited GPU memory size will be elaborated shortly. Second, we identify the edges whose starting vertex and ending vertex are in the set of extracted vertices. The extracted vertices and corresponding selected edges form the subgraph G', which will load to GPU memory to support efficient graph processing.



(a) The original graph (b) The reordered graph

Figure 6: The heatmaps of the edge distribution

We determine the number of the vertices to be extracted at the first step and then fit the extracted subgraph into size-constrained GPU global memory. We first accumulate the total out-going neighbors of the vertices from the ordered list. Second, we binary search the foremost vertex of which the the total size of all of their out-neighbors can be fit into the size-constrained GPU memory.

5 Graph Processing Phase in CGgraph

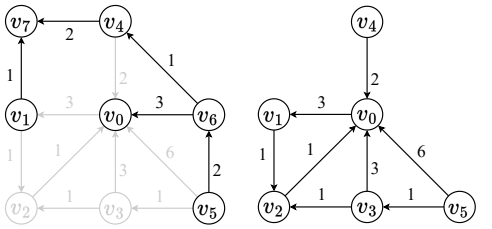
In this section, we overcome the second challenge: *how to cooperate CPU and GPU effectively for fast graph processing?* In particular, we first propose a CPU-GPU cooperative computation scheme for fast graph processing on a single machine with CPU-GPU co-processor in Section 5.1. Next, we devise the processing optimization techniques on CPU and GPU, which fully unlock their computation capabilities, in Sections 5.2 and 5.3, respectively. Last, we present a simple-yet-effective strategy to decide whether CGgraph invokes GPU or not at the beginning of each iteration in Section 5.4.

5.1 CPU and GPU Cooperative Processing

After the data processing phase in Section 4, the graph G is reordered and the subgraph G' which includes those selected high in-degree vertices (and their corresponding selected edges) are loaded into the GPU global memory. It is not trivial to parallel execute the graph processing algorithms via the vertex-centric computation model under the above setting. The major reason is that the subgraph G', which stores in GPU memory, probably only includes a subset of edges for every vertex in it.

In this section, we propose a CPU-GPU cooperative computation scheme to address it. For each iteration, every vertex in graph G is classified into two categories: (i) CPU-only and (ii) CPU-GPU. In particular, if the vertex v is not in subgraph G', i.e., it is not in the GPU memory, it only can be processed on the CPU, thus, vertex v is in the CPU-only category. Otherwise, vertex v is in the CPU-GPU category. It is easy to process the CPU-only vertices via the vertex-centric computation model. Next, we present our CPU-GPU cooperative processing scheme for the CPU-GPU category vertices. For each CPU-GPU vertex v, two computation tasks of it will be instantiated. One of its computation tasks will process a subset of its edges (which is included in G') on GPU, the other computation task of it will process the rest of its edges in the residual graph, i.e., $G - G'$ on CPU. After that, the computed results of both tasks will be aggregated to derive the final result of vertex v.

For implementation-wise, the last aggregation step of every CPU-GPU vertex will do in a batch at the end of each iteration. In particular, every vertex value of GPU-processed computation tasks



(a) Graph G on CPU (b) Subgraph G' on GPU

Iteration	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
	CPU/GPU	CPU/GPU	CPU/GPU	CPU/GPU	CPU/GPU	CPU/GPU	CPU-only	CPU-only
init	+∞/+∞	+∞/+∞	+∞/+∞	+∞/+∞	+∞/+∞	0	+∞/–	+∞/+∞
1st	+∞/6	+∞/+∞	+∞/+∞	+∞/1	+∞/+∞	0	2/–	+∞/–
2nd	5/4	+∞/9	+∞/2	1	3/+∞	0	2	+∞/–
3rd	4/3	9/7	2	1	3	0	2	5/–
4th	3	7/6	2	1	3	0	2	5
5th	3	6	2	1	3	0	2	5

(c) Iterations of SSSP algorithm

Figure 7: Cooperative CPU-GPU computation and its illustrative example of SSSP algorithm

will be moved back to the CPU main memory together. Each of them will be aggregated with its corresponding vertex value on the CPU main memory to derive the final result. In addition, the updated value of every node in subgraph G' will sync to the GPU memory at the beginning of the next iteration to guarantee the GPU computation tasks will not work on the out-of-date values. With the above vertex value synchronization method, the CPU and GPU computation task of the same vertex does not need to be processed at the same iteration. The value movement overhead is quite small as our subgraph extraction approach in Section 4.4 only extracts a subset of vertices from G, we will verify it by experiments in Section 6.

Example. We use the graph G in Figure 7 as an example to illustrate how to cooperate CPU and GPU to process the SSSP algorithm via the vertex-centric computation model. Figure 7(a) is the reordered graph G, the gray part in it is the subgraph G' on GPU memory, see Figure 7(b), which is selected by the data processing phase. The black part is the residual graph, i.e., $G - G'$. To improve the computation efficiency, for the CPU-GPU co-processing vertex v, CGgraph only processes its edges on the residual graph $G - G'$ via its computation task on the CPU.

Given the source node v_5, Figure 7 (c) shows the SSSP algorithm processing procedure. The computation tasks of v_6 and v_7 are CPU-only and the rest vertices in G are in the CPU-GPU category, as the first row in Figure 7(c) shown. The distance of the source vertex is initialized to 0 and others are set to +∞ in CPU memory at first. During the first iteration, the initialized vertex values are copied to GPU memory before computation task execution. The vertex value, i.e., the shortest distance of it found so far, of vertex v_6, is updated by CPU-only computation task as v_6 is not in the subgraph G'. The vertex value of v_0 and v_3 are updated by their corresponding GPU-based computation task, and the values are moved to the CPU main memory for final aggregation. For example, the distance value of v_0 and v_3 on the CPU main memory are updated to 6 and 1, respectively, as the red color highlighted values at the 1st iteration row in Figure 7(c). Moreover, the vertices v_0, v_1, v_2, and v_4 are active and waiting for processing at the next iteration. For the second iteration, the updated vertex values of v_0 to v_5 are moved to GPU global memory at first. The distance values of v_1 and v_2 are updated by their corresponding GPU computation tasks. However, the distance value of v_4 is updated by its CPU computation task as there is an edge (v_5, v_4) in the residual graph. For CPU-GPU vertex v_0, its CPU computation task updates its value to 5 via the path $v_5 \rightarrow v_6 \rightarrow v_0$, and its GPU computation task updates its value to 4

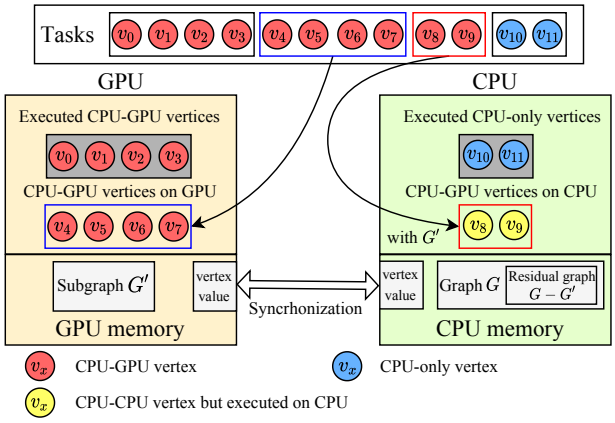


Figure 8: On-demand task allocation

via the path $v_5 \rightarrow v_3 \rightarrow v_0$. The final distance value of v_0 is obtained by aggregating the results of CPU and GPU computation tasks, i.e., $\min\{5, 4\} = 4$, as red color highlighted. CGgraph executes the SSSP algorithm by following the above steps in each iteration until all the distance values do not change anymore.

On-demand task allocation. Even though the above CPU and GPU cooperative processing scheme is very efficient, it still has room for further performance improvement. Specifically, the vertices in the CPU-GPU category (i.e., in subgraph G') will be co-processed via CPU and GPU by default. However, it may incur the workload imbalance between the CPU and GPU if there are too many vertices waiting for execution on the GPU. In this section, we alleviate the workload imbalance issue by proposing the following on-demand task allocation approach.

Figure 8 shows our on-demand task allocation approach for CPU-GPU co-processing. At the beginning of each iteration, the vertices will be classified into CPU-only (in blue color) and CPU-GPU (in red color) categories, as the task box in Figure 8 shown. We then invoke CPU and GPU to cooperatively process these tasks by the above CPU-GPU cooperative processing scheme. When the CPU executed all these CPU-only vertices, it will start to process the CPU-GPU vertices, which originally are waiting for GPU execution, as the tasks of v_8, v_9 in the red box of Figure 8 shown. Specifically, the CPU-GPU vertices will not be executed by the GPU anymore if they are allocated to the CPU for execution as all their edges can be obtained from the graph G, which is in the CPU's main memory. As illustrated in Figure 8, the computation task of v_8 and v_9 (which

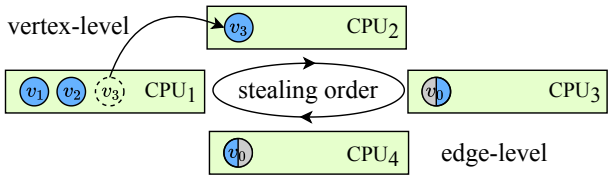


Figure 9: Task processing on CPU

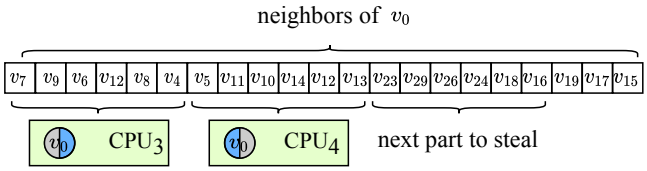


Figure 10: Edge-level work stealing

originally are CPU-GPU vertices in red color) are executed by the CPU with all their edges in G, see the yellow vertices in Figure 8.

Until now, we devise a CPU-GPU cooperative processing scheme with on-demand task allocation approach to cooperate with the CPU and GPU for graph processing in CGgraph. In subsequent, we present the performance optimization techniques for efficient task processing on CPU and GPU in Sections 5.2 and 5.3, respectively.

5.2 Tasks Processing on CPU

To improve the utilization of every CPU core, we devise vertex-level and edge-level work stealing mechanisms in this section.

Vertex-level work stealing. For every vertex in each iteration, their corresponding computation tasks are uniformly assigned to the task queue of each CPU core, and wait for execution. As shown in Figure 9, each CPU core has its own working queue, which includes the assigned tasks in this iteration. Each CPU core executes them one by one from the head to the tail of its working queue. The computation intensiveness of different tasks is quite different, which results in the different completion time for each CPU core. We devise vertex-level work stealing method to address the above load imbalance issue among CPU cores.

Take Figure 9 as an example, suppose there are three computation tasks in CPU_1 (i.e, v_1, v_2 and v_3) when CPU_2 completes. Then, CPU_2 will steal tasks from the tail of CPU_1's working queue, i.e., the computation task of v_3. We impose a clockwise stealing order to guarantee the correctness and efficiency of execution, e.g., CPU_2 will steal the tasks in the working queue of CPU_1 first, it will steal the task of CPU_4 if and only if it cannot steal tasks from CPU_1. In our implementation, the CPU core will steal 64 tasks at a time as we use a bitset with 64 bits to filter the inactive vertices.

Edge-level work stealing. The execution time of computation tasks in each iteration always follows the long-tail distribution, e.g., a few vertices incur a very long execution time. To alleviate the load imbalance issue caused by a few long computation tasks, we further provide a fine-grained work stealing method on the CPU, i.e., edge-level work stealing. For example, the CPU_4 steals partial work of task with vertex v_0, which is executing by CPU_3, as shown in Figure 9. We adapt the work splitting strategy [15] to split a vertex that has a large number of neighbors. In particular, when the CPU core processes a task associated with a large number of

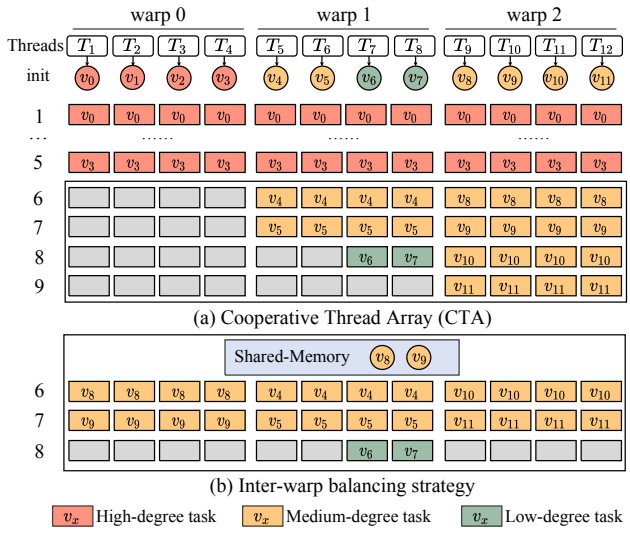


Figure 11: Task processing on GPU: CTA vs. IWB

edges, i.e., it exceeds a threshold ϵ, we will mark it as "splittable". It means that the task can be stolen partially by other CPU cores, we use the same stealing order as the vertex-level stealing method. For a "splittable" task, each CPU core only processes a subset of edges when processing it, and the rest edges can be processed by other CPU cores. For example, as shown in Figure 10, the CPU_3 processes the first 8 edges when it executes the task with v_0, and the CPU_4 steals the task v_0 with the second 8 edges and executes it. Therefore, multiple CPU cores work together to complete the computation task of v_0.

In summary, for all the computation tasks on the CPU, we first employ the vertex-level work stealing method to balance the workload among all CPU cores. After that, the edge-level work stealing method is applied to reduce the execution time of the tasks which have a large number of neighbors.

5.3 Tasks Processing on GPU

We exploit the computation ability of GPU by reducing the warp divergence during computation task processing on it. In the literature, cooperative thread array (CTA) [9, 46, 48] is a widely-used method to execute the computation tasks on GPU. It processes 256 computation tasks in a batch. CTA executes the high-degree tasks (i.e., the corresponding vertices have more than 256 edges) in a block manner, i.e., all 256 threads in a block will execute the task together. The medium-degree tasks, which edges are in the range of 32 to 256, will be executed in a warp manner, i.e., use 32 threads in a warp to execute it. The low-degree tasks (i.e., less than 32 edges of their vertices) will be executed by a single thread.

Figure 11(a) depicts the task processing procedure example of CTA for a batch of computation tasks. It executes high-degree tasks (i.e., tasks of v_0, v_1, v_2, v_3) by employing all threads across all warps in the block. The medium-degree tasks will be executed by all threads within a warp. For example, tasks of v_4, v_5 and tasks of v_8, v_9, v_{10}, v_{11} are processed by the threads in warp_1 and warp_2, respectively. The low-degree tasks are executed by a single thread, see tasks of v_6 and v_7. CTA achieves high GPU utilization via different

Table 2: The statistics of the used graph datasets

Graph	$\|V\|$	$\|E\|$	D_{avg}	Sink	Size	Domain
gsh2015tpd (GS)	31M	581M	18.8	24.9%	5GB	Web
twitter2010 (TW)	42M	1,469M	35.2	4.5%	12GB	Social
friendster (FR)	124M	1,806M	14.56	18.1%	15GB	Social
weibo (WB)	70M	2,586M	37.1	14.6%	21GB	Social
uk-2006 (UK)	78M	2,965M	38.01	11.3%	25GB	Web
uk-union (UN)	134M	5,505M	41.2	10.1%	50GB	Web

execution manners for different tasks. However, it incurs significant inter-warp divergence when processing medium-degree tasks. For example, the threads in warp 0 are stalled after they processed the task with v_3.

To address the inter-warp divergence on GPU, we re-assign the medium-degree tasks from overloaded warps to others. In particular, we calculate the average medium degree among warps. If the medium-degree tasks in a warp are larger than the average, we move the over part to the shared memory. For example, the average number of medium-degree tasks is 2 (6 tasks and 3 warps). Thus, the overloaded warp 2 will move two tasks (i.e., tasks of v_8 and v_9) to shared memory. Thus, both tasks can be executed by the threads in warp 0 after the processing of the task with v_3, see Figure 11(b). In the implementation, we guarantee the number of tasks in the shared memory will not exceed the size of it. In addition, we apply the warp appending method [46] to visit the shared memory to reduce the number of atomic operations.

5.4 GPU Invoking Strategy

Obviously, invoking GPU for cooperating incurs extra costs. If the GPU invoking overhead is larger than the performance improvement we can get by exploiting CPU-GPU cooperative processing, then we do not need to invoke it. As the profiling result shown in Figure 1(a), the number of active edges at the beginning or ending iterations is quite small. For example, the number of active edges in the first 2 iterations and the last 10 iterations are less than 1,500. Intuitively, invoking GPU in these iterations is not a good choice as it cannot fully exploit the high-parallelism of GPU but incurs extra overhead. In CGgraph, we propose a simple-yet-effective GPU invoking strategy to decide whether it invokes GPU or not at the beginning of each iteration. In particular, the GPU processor only be invoked if and only if the number of active edges in the iteration is larger than the given threshold τ and the percentage of the active edges in subgraph G' is larger than the given threshold θ. The threshold parameters τ and θ in CGgraph are tunable for different datasets and different graph processing algorithms.

6 Experimental Evaluation

We evaluate the superiority of our proposed CGgraph via extensive experimental evaluation in this section. In Section 6.1, we describe the experimental setting. We present the findings from the experimental results of overall performance evaluation in Section 6.2. In Section 6.3, we investigate the effects of our designed techniques in CGgraph. We last conduct performance evaluation experiments with a high-end GPU server in Section 6.4.

6.1 Experimental Setting

Graph dataset. Table 2 shows the statistics of six real-world graph datasets, which are used in the evaluation. The scale of the graphs ranges from 5GB to 50GB. By following the setting of [39], we remove duplicate edges and self-loops from all graph datasets before graph processing. All these graph processing systems can process both directed and undirected graphs, however, the undirected graphs could be converted to directed ones by replacing each undirected edge with a pair of directed edges. Thus, all edges are directed in all experiments. The percentage of sink nodes in these graphs is from 4.5% to 24.0%.

State-of-the-art systems. We compared CGgraph with 7 state-of-the-art graph processing systems in 3 categories, i.e., CPU-only, GPU-only, and CPU-GPU heterogeneous systems:

- Galois(V4.0) [35]. It is a CPU-only graph processing system on a single machine. It is a state-of-the-art CPU-only system, we use the source code provided by its github repository [2].
- Gemini [55]. It is a distributed CPU-only graph processing system. It has notable shared-machine performance. To provide a fair comparison, we remove its distributed design (e.g., messaging abstraction) from its source code [3].
- Gunrock(V2.0) [48]. It is a GPU-only graph processing system that achieves good performance via its novel frontier-based approach. The source code is available at [5].
- Groute [9]. It is a GPU-only graph processing system, it enjoys excellent performance via its GPU-based asynchronous execution and communication. Its github repository is at [4].
- Totem [18]. It is a CPU-GPU heterogeneous graph processing system. It statically partitions the graph into two subgraphs, one is used by the CPU and the other is loaded into GPU global memory and used by the GPU. The updated vertex values of each subgraph are synchronized after each iteration. The source code is available at [8].
- Subway [38]. It is also a CPU-GPU heterogeneous graph processing system. It generates a subgraph that contains only active edges and vertices on GPU memory. To further improve the performance, it reduces both the number of times the subgraph is loaded and the size of the loaded subgraph each time. We obtain its source code at [7].
- LargeGraph [52]. It is a CPU-GPU heterogeneous graph processing system, which uses a dependency-aware data-driven execution approach to achieve good performance. We cannot obtain the source code from the authors of [52], thus, we tried our best to re-implement it by ourselves, which is open-sourced at [6].

Graph algorithms. We employ 4 representative graph algorithms: breadth-first search (BFS), single-source shortest path (SSSP), weakly connected component (WCC), and PageRank (PR) in our evaluation. For BFS and SSSP, we use the same source vertex when comparing with other systems, which are randomly selected from the vertex set with the out-degree not equal to zero. Since SSSP requires graphs with edge weights, we randomly assign weight values between

Table 3: The execution time (in seconds) of compared systems

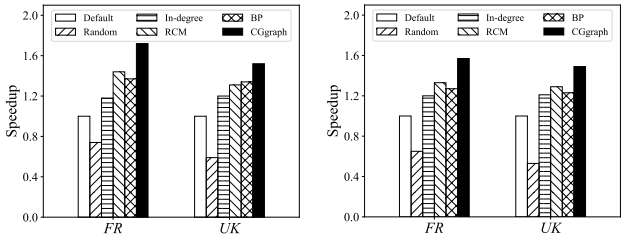
Alg	G	CPU-only		GPU-only		CPU-GPU heterogeneous system			
		Galois	Gemini	Gunrock	Groute	Totem	Subway	LargeGraph	CGgraph
BFS	GS	0.93	1.01	**0.32**	0.57	0.89	0.81	0.73	0.43
	TW	2.24	2.08	-	1.01	2.13	1.89	1.53	**0.66**
	FR	5.20	4.06	-	-	3.05	5.21	4.13	**1.49**
	WB	6.97	6.43	-	-	5.01	6.19	5.12	**2.37**
	UK	5.15	6.03	-	-	4.74	4.79	**1.39**	1.45
	UN	10.42	11.85	-	-	-	-	5.39	**4.15**
SSSP	GS	3.42	3.33	0.94	1.59	2.47	2.01	1.13	**0.77**
	TW	11.81	9.23	-	-	5.50	9.25	8.13	**2.52**
	FR	40.96	31.19	-	-	17.73	16.30	15.27	**3.43**
	WB	52.73	48.67	-	-	-	23.50	21.24	**8.71**
	UK	28.63	31.13	-	-	-	-	5.39	**4.15**
	UN	39.75	-	-	-	-	-	31.33	**15.21**
WCC	GS	2.97	3.73	0.81	**0.77**	1.81	1.73	1.34	0.79
	TW	6.41	8.14	-	3.20	5.97	5.73	5.35	**1.59**
	FR	26.35	22.21	-	-	8.14	10.28	8.76	**2.38**
	WB	37.57	32.13	-	-	14.73	16.32	14.11	**5.01**
	UK	15.13	29.42	-	-	17.81	20.37	11.92	**7.54**
	UN	25.72	35.95	-	-	-	-	18.43	**12.71**
PR	GS	13.72	12.57	3.07	4.56	5.42	4.07	5.49	**2.46**
	TW	58.14	44.83	-	16.12	25.46	23.13	20.50	**5.97**
	FR	92.70	75.37	-	-	41.23	33.73	31.46	**11.83**
	WB	118.59	98.27	-	-	48.27	44.73	45.82	**18.68**
	UK	57.82	38.51	-	-	23.49	20.57	19.27	**8.30**
	UN	69.21	54.61	-	-	-	-	44.26	**24.75**

(0, 100] to every edge by following the setting of the existing systems [55]. For PR, we use the same terminal condition with 0.85 as the damping factor and run it for 20 iterations. The reported measurement is the average of processing the algorithm 10 times.

Implementation and hardware configurations. CGgraph is implemented by over 8000 lines of C++ and CUDA code, compiled with GCC 11.3 and CUDA 11.7 on Ubuntu 18.04 with O3-level optimization. The threshold parameters τ and θ in GPU invoking strategy are 2.5 millions and 35%, respectively. We compare CGgraph with other 7 systems on a modern commodity machine with CPU-GPU co-processor by default. On the host, there is an Intel i9-9900k CPU with 8 cores and 128GB memory. On the device, there is an NVIDIA TITAN Xp GPU with 3840 CUDA cores and 12GB GDDR5 global memory. GPU connects with the host via the PCI Express 3.0 at 16x, which bandwidth is approximately 11 GB/s.

6.2 Overall Performance Evaluation

Table 3 presents the end-to-end processing time (in seconds) of all evaluated systems when processing 4 different algorithms on 6 graph datasets. Firstly, it is no doubt that CGgraph performs the best among all compared systems. In particular, CGgraph ranks 1st among all 8 systems in 21 over 24 test cases, see bold values in the last column of Table 3. In addition, it ranks 2nd in the rest three test cases. However, the performance of CGgraph are comparable with the best in these three cases. Secondly, the GPU-only systems (i.e., Gunrock and Groute) fail to process large graphs as they encountered out-of-memory errors during graph processing, which are indicated with "-" in Table 3. Moreover, the CPU-GPU heterogeneous graph processing systems outperform the CPU-only systems due to the utilization of high parallelism on the GPU. Thirdly, there is no clear winner among the existing CPU-GPU heterogeneous systems (i.e., Totem, Subway, and LargeGraph). Moreover, Totem and Subway cannot process the largest dataset UN in all tested algorithms as they encounter out-of-memory errors. Nevertheless, CGgraph outperforms all of them in almost all tested cases. The core reason is CGgraph addresses GPU memory over-subscription via extracting



(a) SSSP (b) PR

Figure 12: Comparison of different graph reordering methods

a subgraph that only load once but can be used by multiple iterations, and it also devises efficient CPU-GPU cooperative processing scheme, which are equipped optimization techniques for task processing on CPU and GPU. Fourthly, from the graph algorithm-wise, the speedup times of CGgraph over all other systems in SSSP (up to 11.94X) and PR (up to 9.73X) are significantly larger than its in BFS (up to 4.15X). It is because of the algorithmic-logic of SSSP and PR are more complex than BFS, which also indicates the superiority of CGgraph to process complex algorithms on large-scale graphs. Lastly, CGgraph is obviously better than other systems when processing social network graphs (e.g., TW, FR and WB) as the number of computation tasks in these graphs are extremely large in some iterations and the efficient cooperative processing scheme in CGgraph could balance the workload in both vertex-level and edge-level on CPU and GPU.

6.3 Effect of Designed Techniques

In subsequent, we investigate the effect of the designed techniques and optimizations one by one in CGgraph.

6.3.1 *Effect of Graph Reordering*

In this section, we evaluate the effect of the proposed graph reordering algorithm (see Section 4.3) in CGgraph. In particular, we first verify the effect of different graph reordering algorithms on the end-to-end performance of CGgraph, then we demonstrate the performance improved by adapting our graph reordering algorithm to other graph processing systems.

Comparison among different graph reordering algorithms. In this experiment, we compare the following five graph reordering algorithms.

- Default, it uses the default vertices order in the input graph and did not apply any graph reordering algorithm on it in CGgraph.
- Random, we follow other graph systems [17, 49] to randomly shuffle the vertices in the input graph and use the shuffled vertices array in CGgraph.
- In-degree [16], it sorts the vertices in the input graph by the descending order of their in-degree.
- BP [14], the vertices in the input graph are reordered in a compression-friendly manner via recursive graph bisection.
- RCM [13], it employs a BFS-based algorithm to permute a sparse adjacency matrix by reducing its bandwidth.
- CGgraph, it is our proposal, which utilizes the proposed graph reordering algorithm in Section 4.3.

Table 4: The speedup times via CGgraph's graph reordering

System		Galois	Gemini	Totem	Subway	LargeGraph
SSSP	FR	3.01X	2.85X	2.37X	2.26X	2.31X
	UK	3.48X	3.01X	-	2.38X	1.49X
PR	FR	4.66X	3.86X	2.45X	2.24X	2.07X
	UK	4.01X	3.13X	2.19X	2.01X	1.82X

Figure 12 depicts the speedup times of different graph reordering algorithms over the default vertices order in CGgraph. We report SSSP and PR on two different datasets FR and UK and omit others due to page limits. The speedup times of CGgraph over the Default of SSSP and PR on FR are 1.72 and 1.57, and on UK are 1.52 and 1.49, respectively. It consistently outperforms all alternative options because our graph reordering algorithm in CGgraph takes both the in-degree of vertices and the data access locality into consideration. In addition, the Random order performs even worse than Default as it does not exhibit the data access locality in the original graph.

Performance improvement of other systems. In this experiment, we evaluate the generality of our proposed graph reordering algorithm by using the reordered graph in existing 5 graph processing systems, i.e., Galois, Gemini, Totem, Subway and LargeGraph. Gunrock and Groute are ignored as they cannot process large graphs as FR and UK. More specifically, we only change the input graph of these systems and do not revise any execution logic in these existing systems.

Since Totem fails to process SSSP on the original UK (see the corresponding "-" in Table 3), it is not surprising it also fails with the reordered UK (in Table 4). Except it, our graph reordering algorithm improves the performance of all these 5 systems in all tested cases, as shown in Table 4. The overall speedup times among these systems range from 1.82X to 4.66X. Interestingly, even in the recently proposed CPU-GPU heterogeneous graph processing systems (i.e., Subway and LargeGraph), our reordering algorithm brings up to 2.38X speedup in the tested cases. The core reason is our graph reordering algorithms offer excellent data access locality, which significantly reduces cache misses during graph processing.

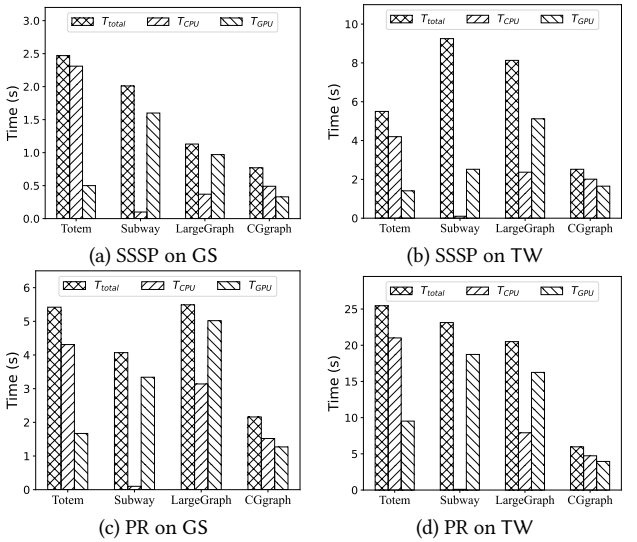
6.3.2 *Effect of Subgraph Extraction Approach*

We then evaluate the effect of the subgraph extraction approach in this section. In particular, there are three approaches for extracting a subgraph from the reordered graph. The first approach is randomly extracting a subset of vertices and their corresponding edges to form the subgraph, we use it as the baseline method. The second approach is extracting the vertices and all of their neighbors as much as possible from the reordered vertices list, we refer to it as *vertex*-based method. The third approach is using our size-constrained subgraph extraction approch in Section 4.4, we refer it as *vertex-edge*-based method.

Table 5 reports the speedup times of *vertex*-based and *vertex-edge*-based approaches over the baseline method when processing SSSP and PR on FR and UK. Specifically, both end-to-end processing time and vertex value synchronization cost are reported. The synchronization cost is updating the vertex values between GPU memory and CPU memory (see Figure 8), which includes both the PCIe cost and the value updating cost in the CPU. First, both *vertex*-based and *vertex-edge*-based approaches are better than the baseline approach, it confirms the effectiveness of our idea to extract a subgraph that

Table 5: The speedup times of subgraph extraction approach

Algorithm	Graph	End-to-end		Synchronization	
		vertex	*vertex-edge*	*vertex*	*vertex-edge*
SSSP	FR	1.51X	1.67X	1.01X	1.21X
	UK	1.50X	1.74X	1.00X	1.69X
PR	FR	1.33X	1.45X	1.00X	1.02X
	UK	1.30X	1.46X	1.00X	1.14X



(a) SSSP on GS (b) SSSP on TW

(c) PR on GS (d) PR on TW

Figure 13: Breakdown of graph processing time

can be used in multiple iterations. Second, the *vertex-edge*-based approach achieves 1.02X to 1.69X improvements on synchronization cost, which is higher than the *vertex*-based approach. The reason is the number of synchronized values in *vertex-edge*-based approach is smaller than the *vertex*-based approach as only a subset of edges of every selected vertex will be included in the extracted subgraph in our approach.

6.3.3 *Effect of CPU-GPU Cooperative Processing*

We next evaluate the effectiveness of our CPU-GPU cooperative processing scheme in CGgraph. We run SSSP and PR on all these 4 CPU-GPU heterogeneous systems. In this experiment, we use the two smallest datasets, i.e., GS and TW, as existing CPU-GPU systems fail to process large graphs in some tested cases. We measure the end-to-end processing time (T_{total}), CPU processing time (T_{CPU}) and GPU processing time (T_{GPU}), and plot them in Figure 13.

First of all, the end-to-end processing time of CGgraph is the smallest among all those 4 systems in every tested case, as T_{total} bar shown in Figure 13. Secondly, the gap between the CPU processing time (T_{CPU}) and the GPU processing time (T_{GPU}) of CGgraph is significantly smaller than the rest 3 systems as we explicitly consider the workload balance between CPU and GPU via on-demand task allocation in Section 5.1. Thirdly, the end-to-end processing time (T_{total}) of Subway and LargeGraph in all tested cases are obviously larger than their CPU processing time (T_{CPU}) as both systems do not balance the workload on CPU and GPU explicitly. Additionally, the data movement or the subgraph generation in them incurs extra overhead.

Table 6: Comparison of workload balance methods on CPU

Algorithm	Graph	V-Donating	V-Stealing	V/E-Stealing
SSSP	FR	1.11X	1.33X	1.41X
	UK	1.17X	1.21X	1.33X
PR	FR	1.12X	1.30X	1.36X
	UK	1.10X	1.35X	1.44X

Table 7: Comparison between CTA and IWB

Algorithm	Graph	L1 Hit ratio		AT/warp		Speedup
		CTA	IWB	CTA	IWB	
SSSP	GS	41.2%	65.7%	19.14	22.86	1.30X
	TW	33.3%	56.8%	21.18	22.59	1.44X
PR	GS	45.9%	70.7%	19.41	23.63	1.33X
	TW	44.9%	66.6%	24.20	25.98	1.38X

6.3.4 *Effect of Optimizations on CPU*

In this section, we verify the effectiveness of our proposed optimization techniques on the CPU. The baseline idea to process the tasks on the CPU is equally dividing them into the task queue of every core on the CPU, then each core exclusively executes the assigned tasks in its task queue. To improve the load balance among different cores in the CPU, there are three methods have been explored.

- V-Donating [23]. An overloaded core shares its tasks with underloaded cores at the vertex-level.
- V-Stealing [3]. An underloaded core steals tasks from overloaded cores at the vertex-level.
- V/E-Stealing. Our proposed solution in Section 5.2, which allows the underloaded core to steal tasks from overloaded cores at both vertex-level and edge-level.

Table 6 shows the speedup times of different methods over the baseline by running SSSP and PR on FR and UK. It is no doubt all the above three CPU load balance solutions achieve better performance when compared with the baseline method, which does not balance the workload among different cores on the CPU at all. Even both V-Stealing and V-Donating methods balance the workload at vertex-level, V-Stealing is slightly better than V-Donating as V-Donating incurs extra cost to collect the status of other cores and write shared tasks into shared memory. Last but the most important, our proposed V/E-Stealing performs the best among all these methods. The key reason is that it guarantees the fine-grained workload balance, i.e., both vertex-level and edge-level, of different cores on the CPU.

6.3.5 *Effect of Optimizations on GPU*

We last evaluate the impact of our proposed Inter-Warp balancing (IWB) optimization for task processing on GPU. In particular, we compare the performance of IWB with the widely-used CTA by running SSSP and PR on GS and TW. We measure and report the following three metrics [45, 47]: (i) L1 cache hit ratio, (ii) the average active threads per warp (denoted as AT/Warp), and (iii) the end-to-end speedup times of CGgraph with IWB over CGgraph with CTA.

As depicted in Table 7, IWB not only achieves high GPU utilization (as demonstrated by AT/WARP) but also increases the L1 cache hit ratio. Since IWB allows the idle threads to fetch tasks to run from the shared memory, the average activated threads in the warp of IWB is larger than it is of CTA. The L1 cache hit ratio of IWB is higher than its of CTA, e.g., it ranges from 56.8% to 70.7% in all tested cases with IWB, but it is only 33.3% to 45.9% with CTA. This is because IWB explicitly moves the medium-degree vertices to L1 cache. The high L1 cache hit ratio and more running threads per warp lead to the speedup times of IWB over CTA ranging from 1.33X to 1.44X.

Table 8: Performance evaluation on a high-end server

G	SSSP			WCC			PR		
	HS-BoE	SM-CGg	HS-CGg	HS-BoE	SM-CGg	HS-CGg	HS-BoE	SM-CGg	HS-CGg
GS	0.40	0.77	0.55	0.45	0.79	0.52	1.59	2.46	1.73
TW	2.10	2.52	1.57	1.11	1.59	1.05	4.27	5.97	3.65
FR	10.27	**3.43**	2.07	4.73	**2.38**	1.45	18.63	**11.83**	9.43
WB	14.49	**8.71**	4.79	5.26	**5.01**	3.11	27.75	**18.68**	14.13
UK	7.54	**4.15**	3.89	4.51	7.54	5.31	13.76	**8.30**	5.46
UN	21.18	**15.21**	9.27	13.07	**12.71**	8.99	28.39	**24.75**	15.45

6.4 Performance Evaluation on High-end Server

We evaluate CGgraph and other systems on a high-end CPU-GPU server, which was equipped with a 12-core Intel Xeon Silver 4214 CPU and an NVIDIA Tesla V100S GPU featuring 5120 cores and 32GB of global memory. It is worth to point out that the size of largest dataset UN is obvious larger than the global memory size of the high-end CPU-GPU server. Specifically, we compare the best performance of existing systems on the high-end server (i.e., HS-BoE), the performance of CGgraph on a single modern commodity machine (i.e., SM-CGg), and the performance of CGgraph on the high-end server (i.e., HS-CGg). In Table 8, we show the experimental results by running SSSP, WCC and PR in these 7 datasets, and refer the interested readers to our technical report [1] for complete experimental results. The key findings are two-fold: (i) CGgraph consistently outperforms existing systems with the high-end server, see the values in the columns of HS-CGg in Table 8; (ii) the performance of our CGgraph on modern commodity CPU-GPU co-processor is better than (see bold values in Table 8) or at least comparable with the best performance of other systems on high-end servers. For example, the best performance of existing systems to process SSSP on UN with a high-end server is achieved by LargeGraph, which takes 21.18s. However, CGgraph only takes 15.21s to process SSSP on UN on a modern commodity machine.

7 Conclusion

In this paper, we propose an ultra-fast graph processing system CGgraph on a modern commodity machine with the CPU-GPU co-processor. CGgraph achieves significant speedup by overcoming GPU memory over-subscription and providing an efficient CPU-GPU cooperative processing scheme. First, CGgraph follows the principle of "load once and multiple times" to extract a subgraph and load it into GPU global memory. Then, CGgraph proposes on-demand task allocation to process graph algorithm by efficient CPU-GPU cooperative processing scheme. Furthermore, both the edge-level and vertex-level workload balance have been explicitly solved in it. In the evaluation, we compare CGgraph with 7 state-of-the-art graph processing systems in various experimental settings. The results indicate that CGgraph achieves a substantial performance improvement on a commodity CPU-GPU co-processor. In the future, we plan to study the distributed CPU-GPU heterogeneous graph processing system.

References

[1] 2023. *CGgraph (Technical Report).* https://github.com/PengBo410/CGgraph/blob/main/CGgraph/CGgraphTR.pdf

[2] 2023. *Galois.* https://github.com/IntelligentSoftwareSystems/Galois

[3] 2023. *Gemini.* https://github.com/thu-pacman/GeminiGraph

[4] 2023. *Groute.* https://github.com/groute/groute

[5] 2023. *Gunrock.* https://github.com/gunrock/gunrock

[6] 2023. *Our LargeGraph Implementation.* https://github.com/PengBo410/CGgraph/blob/main/CGgraph/lg/

[7] 2023. *Subway.* https://github.com/AutomataLab/Subway

[8] 2023. *Totem.* https://github.com/netsyslab/Totem

[9] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. *ACM SIGPLAN Notices* 52, 8 (2017), 235–248.

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

[11] Jie Chen and Yousef Saad. 2010. Dense subgraph extraction with application to community detection. *TKDE* 24, 7 (2010), 1216–1230.

[12] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, and Sriram Sankar. 2013. Unicorn: A system for searching the social graph. *PVLDB* 6, 11 (2013), 1150–1161.

[13] Elizabeth Cuthill and James McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *ACM National Conference*. 157–172.

[14] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing graphs and indexes with recursive graph bisection. In *SIGKDD*. 1535–1544.

[15] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. In *SC*. 1–11.

[16] Priyank Faldu, Jeff Diamond, and Boris Grot. 2019. A closer look at lightweight graph reordering. In *IISWC*. IEEE, 1–13.

[17] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing large graphs on GPUs with unified memory. *PVLDB* 13, 7 (2020), 1119–1133.

[18] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT*. 345–354.

[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*. 17–30.

[20] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*. 599–613.

[21] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *PACT*. 233–245.

[22] Hideya Iwasaki, Kento Emoto, Akimasa Morihata, Kiminori Matsuzaki, and Zhenjiang Hu. 2022. Fregel: a functional domain-specific language for vertex-centric large-scale graph processing. *Journal of Functional Programming* 32 (2022), e4.

[23] Joseph John, Josh Milthorpe, and Peter Strazdins. 2022. Distributed Work Stealing in a Task-Based Dataflow Runtime. In *PPAM*. Springer, 225–236.

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

[25] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *SIGMOD*. 447–461.

[26] Jon Kleinberg and Eva Tardos. 2006. *Algorithm design.* Pearson Education India.

[27] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2020. Gpop: A scalable cache-and memory-efficient framework for graph processing over parts. *TOPC* 7, 1 (2020), 1–24.

[28] Victor E Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. 2010. A survey of algorithms for dense subgraph discovery. *Managing and mining graph data* (2010), 303–336.

[29] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A framework for memory oversubscription management in graphics processing units. In *ASPLOS*. 49–63.

[30] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *SC*. 1–12.

[31] Haotian Liu, Bo Tang, Jiashu Zhang, Yangshen Deng, Xiao Yan, Xinying Zheng, Qiaomu Shen, Dan Zeng, Zunyao Mao, Chaozu Zhang, et al. 2022. GHive: accelerating analytical query processing in apache hive via CPU-GPU heterogeneous computing. In *SoCC*. 158–172.

[32] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146.

[33] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *PPoPP*. 201–213.

[34] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM SIGPLAN Notices* 47, 8 (2012), 117–128.

[35] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.

[36] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *PVLDB* 11, 12 (2018), 1876–1888.

[37] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal* 25 (2016), 125–150.

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

[39] Peter Sanders, Christian Schulz, and Dorothea Wagner. 2014. Benchmarking for graph clustering and partitioning. *ESNAM* (2014).

[40] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. Graphreduce: processing large-scale graphs on accelerator-based systems. In *SC*. 1–12.

[41] Xuanhua Shi, Xuan Luo, Junling Liang, Peng Zhao, Sheng Di, Bingsheng He, and Hai Jin. 2017. Frog: Asynchronous graph processing on GPU with hybrid coloring model. *TKDE* 30, 1 (2017), 29–42.

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

[43] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings 20*. Springer, 451–462.

[44] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From" think like a vertex" to" think like a graph". *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[45] Charbel Toumieh and Alain Lambert. 2021. Gpu accelerated voxel grid generation for fast mav exploration. *arXiv:2112.13169* (2021).

[46] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *PPoPP*. 38–52.

[47] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. Grus: Toward unified-memory-efficient high-performance graph processing on gpu. *TACO* 18, 2 (2021), 1–25.

[48] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *PPoPP*. 1–12.

[49] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *SIGMOD*. 1813–1828.

[50] Ye Yuan, Guoren Wang, Lei Chen, and Haixun Wang. 2013. Efficient keyword search on uncertain graph data. *TKDE* 25, 12 (2013), 2767–2779.

[51] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *Big Data*. IEEE, 293–302.

[52] Yu Zhang, Da Peng, Xiaofei Liao, Hai Jin, Haikun Liu, Lin Gu, and Bingsheng He. 2021. LargeGraph: An efficient dependency-aware GPU-accelerated large-scale graph processing. *TACO* 18, 4 (2021), 1–24.

[53] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *PACMPL* 2, OOPSLA (2018), 1–30.

[54] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *TPDS* 25, 6 (2013), 1543–1552.

[55] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system.. In *OSDI*, Vol. 16. 301–316.

A Experimental Evaluation on High-end Server

In this section, we evaluate CGgraph on a server to run the the graph datasets and algorithms in Section 6. The server consisted of a 12 cores Intel Xeon silver 4214 CPU, and an NVIDIA Tesla V100S GPU with 5120 cores and 32GB global memory. GPU connected to the host via the PCI Express 3.0 at 16x, which is approximately 11 GB/s.

Table 9: The execution time (in seconds) of compared systems on a high-end server

Alg	G	CPU-only		GPU-only		CPU-GPU heterogeneous system			
		Galois	Gemini	Gunrock	Groute	Totem	Subway	LargeGraph	CGgraph
BFS	GS	0.65	0.70	**0.15**	0.29	0.65	0.61	0.55	0.31
	TW	1.69	1.61	**0.39**	0.66	1.51	1.35	1.17	0.45
	FR	3.77	2.98	-	1.24	2.27	3.69	3.66	**0.98**
	WB	5.08	4.72	-	2.25	2.94	4.12	4.30	**1.52**
	UK	3.54	4.95	-	1.98	3.34	3.31	**1.02**	1.19
	UN	6.85	7.31	-	-	-	-	4.01	**3.07**
SSSP	GS	2.11	2.02	**0.40**	0.99	1.75	1.28	0.80	0.55
	TW	7.33	6.15	2.10	3.21	3.84	4.79	4.61	**1.57**
	FR	25.76	21.07	-	10.27	11.98	10.81	11.39	**2.07**
	WB	32.35	31.81	-	14.49	18.45	19.58	17.13	**4.79**
	UK	18.63	19.83	-	7.54	12.01	15.64	10.81	**3.89**
	UN	25.81	22.44	-	-	-	-	21.18	**9.27**
WCC	GS	2.13	2.65	**0.45**	0.48	1.37	1.32	1.11	0.52
	TW	4.51	5.81	1.11	1.78	3.92	3.84	4.28	**1.05**
	FR	18.95	16.21	-	4.73	5.89	7.61	5.43	**1.45**
	WB	27.03	23.45	-	5.26	9.69	9.73	9.66	**3.11**
	UK	10.01	12.53	-	**4.51**	11.79	8.24	6.62	5.31
	UN	16.70	22.81	-	-	-	-	13.07	**8.99**
PR	GS	8.11	7.39	**1.59**	3.27	3.00	2.76	3.74	1.73
	TW	35.23	26.21	4.27	6.97	14.07	16.29	14.96	**3.65**
	FR	60.119	44.86	-	18.63	23.56	21.21	22.63	**9.43**
	WB	71.87	57.47	-	27.75	29.25	30.47	32.51	**14.13**
	UK	32.48	22.01	-	14.51	15.15	15.08	13.76	**5.46**
	UN	40.47	30.01	-	-	-	-	28.39	**15.45**

Table 9 presents the end-to-end processing time (in seconds) of all evaluated systems when processing 4 different algorithms on 6 graph datasets on the server. We next present three key observations in it as follows. First of all, the overall performance of CGgraph is the best among all these evaluated systems by various experiments on the underlying high-end sever. For instance, for overall 24 tested cases (4 algorithms with 6 datasets), it ranks 1st in 17 cases and 2nd in 5 cases. Secondly, this is no doubt the performance of every system can be improved via the high-end sever when comparing with the corresponding performance on modern commodity machine (see Table 3 in Section 6.2). For example, CGgraph on high-end server is consistently faster than CGgraph on modern commodity machine. Thirdly, the overall performance of CGgraph on modern commodity machine, as depicted in Table 3, is better than (or at least is comparable to) the performance of other systems running on the high-end servers, as shown in Table 9. In particular, CGgraph on modern commodity machine ranks 1st among all 8 systems in 11 over 18 test cases by running SSSP, WCC and PR on the first 6 different datasets. For BFS, CGgraph on modern commodity machine shows comparable performance with these systems on high-end server, the reason is the algorithm-logic of BFS is simple and all these systems are good enough and show similar performance on tested cases.

B The reorder algorithm of CGgraph

n this section, we first present the pseudocode for the graph reordering as Algorithm 1. Secondly, we present the proof of its time complexity. Lastly, we provide the execution times of the algorithm on six distinct datasets.

Algorithm 1: Graph Reorder

Input: the list of all vertices V of the graph
Output: the reordered array W

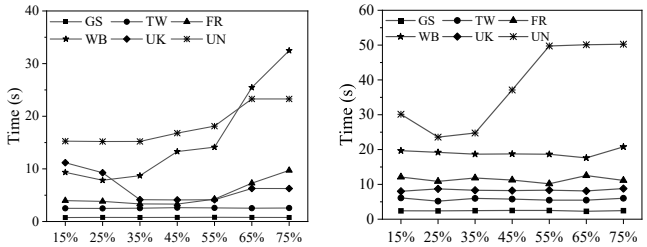
```
1  S ← {} ; // List for storing all sink vertices
2  N ← [] ; // Array for sorting neighbors of a vertex
3  head, tail ← 0;
4  V ← sort the vertices in V by in-degree;
5  foreach vertex v_i ∈ V do
6      if v_i ∉ (W ∪ S) then
7          if v_i has out-neighbors then
8              w[tail++] ← v_i;
9          else
10             append v_i to S;
11         while head < tail do
12             v_cur ← W[head++];
13             N ← all out-neighbors of v_cur;
14             N ← sort vertices in N by in-degree;
15             foreach vertex v_j in N do
16                 if v_j ∉ (W ∪ S) then
17                     if v_j has out-neighbors then
18                         w[tail++] ← v_j;
19                     else
20                         append v_j to S;
21 foreach vertex v_k in S do
22     w[tail++] ← v_k;
23 return W;
```

To achieve our goal, we present a heuristic graph reordering algorithm as shown in Algorithm 1. The algorithm takes all vertices V of the graph as input, and output the reordered vector W. It includes three steps:

First, it sorts elements in V in a descending order based on their in-degree (line 4). We using the standard library's *std::sort* function for sorting in C++, its time complexity is $O(n \cdot logn)$, where n represents the number of the vertices.

Second, it takes a vertex v_i (line 5) from the head of V and adapt a BFS-like algorihm, so that the adjacent vertices will be explored (lines 11-20). Regarding time complexity, the second step consists of four terms (line 5, line 11, line 14 and line 15). The first and second terms are used to obtain the vertex order for performing BFS. They require a total of n iterations, resulting in a time complexity of $O(n)$. In each iteration, the third item sort the s elements in N by their in-degree in $O(d_{max} \cdot log(d_{max}))$, the fourth item scans the out-neighbors of each vertex in $O(d_{max})$, where d_{max} denotes the maximum out-degree of the graph. Therefore, the overall time complexity of the second step is $O(n \cdot d_{max} \cdot log(d_{max}))$.

Third, we append the sink vertices from S to the end of W (lines 21-22) in $O(n)$.



(a) SSSP (b) PR

Figure 14: The performance of SSSP and PR for different θ.

As a result, the total time complexity of the reorder algorithm is $O(n \cdot logn + n \cdot d_{max} \cdot log(d_{max})))$.

Additionally,we measured the time cost of our graph reordering on 6 datastes as follows: GS(32.3s), TW(145.2s), FR(162.3s), WB(212.3s), UK(78.2s) and UN(128.9s).

C Effect of GPU Invoking parameters

In CGgraph, the GPU processor only is invoked if and only if the number of active edges in the iteration is larger than the given threshold τ and the percentage of the active edges in subgraph G' is larger than the given threshold θ. Generally, when τ and θ is small, the GPU will be involved in more iterations, potentially including some iterations where the benefit of its fast computation may not outweigh the launch and data movement overhead. In contrast, when τ and θ is large, the GPU will be involved in fewer iterations, potentially miss some compute-intensive iterations. Because we observed that, in all 6 datasets we used, when the number of active edges in an iteration exceeds a specific threshold, the execution time of that iteration comprises a significant portion of the total execution time when only using the CPU. For the six datasets, these threshold values are 3.15 million, 3.23 million, 2.51 million, 2.92 million, 2.78 million, and 2.85 million, respectively. Therefore, we decided to set the threshold τ to a fixed value of 2.5 million. Figure 14 plots the execution time of SSSP and PR for different θ under the fixed threshold τ of 2.5 million. We observed that when the GPU can load the entire graph, it does not significantly affect the GPU invocation. However, when the GPU only stores a subgraph, both excessively large and small values of θ can reduce overall performance. Therefore, we set θ to 35% default as it seems to work well in all cases.